UNIVERSIDAD AUTÓNOMA DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

DISEÑO E IMPLEMENTACIÓN DE UN SISTEMA DE STREAMING DE MÚSICA

Adrián Bueno Jiménez
Tutor: Álvaro Ortigosa Juárez

Mayo 2017

DISEÑO E IMPLEMENTACIÓN DE UN SISTEMA DE STREAMING DE MÚSICA

AUTOR: Adrián Bueno Jiménez
TUTOR: Álvaro Ortigosa Juárez

Dpto. de Ingeniería Informática
Escuela Politécnica Superior
Universidad Autónoma de Madrid
Mayo de 2017

Resumen

Este Trabajo Fin de Grado tiene como objetivo crear un servicio de streaming de música adaptativo y un cliente web que lo utilice. Con adaptativo quiero decir que la calidad del audio obtenido por las aplicaciones cliente irá variando en función de la calidad de la red.

El servidor contará con una API REST para intercambiar datos entre el servidor y los clientes. Nos permitirá registrar usuarios, subir canciones y crear listas de reproduccion. Con las canciones y las imágenes subidas se crearán varias copias en distintas calidades, el audio se segmentará en pequeños trozos que nos permitirán hacer que los clientes empleen los segmentos con el nivel de calidad adaptado a cada instante dado. Los clientes pueden tener intercalados segmentos de distintas calidades que se reproducirán como un sola canción. Añadiremos una capa de seguridad a nuestro servidor mediante el uso de OAUTH2, que nos permitira autenticar tanto a los usuarios como a los clientes y prevenir que los mismos guarden la contraseña de los usuarios. El servidor sera desarrollado en Node.js y utilizara una base de datos MongoDB para la persistencia de los datos.

El cliente sera una pagina web de una sola pagina hecha con Angular. La pagina sera capaz de adaptarse a cualquier pantalla. Tambien tendra algun componente táctil para que tenga una mejor experiencia en smartphones y tabletas. Al escribirla con Angular tambien podemos llevar nuestra aplicación en un futuro al escritorio o a los moviles haciendo uso de frameworks como Electron o Ionic.

Abstract

The aim of this Bachelor Thesis is to create an adaptative music streaming service and a web client that uses it. Adaptative means that the audio quality obtained by the client applications will vary according to the quality of the network.

The server will have a REST API to exchange data between server and clients. We will be able to register users, upload songs, and create playlists. We will process audio and images uploaded to make several copies of different qualities, audio will be segmented into small pieces that will allow clients to get the best quality piece at every moment. Clients can merge segments of different qualities that will be played as a single track. We will add a layer of security to our server through the use of OAUTH2 that will allowe us to authenticate both, users and clients, and prevent customers from saving users password. The server will be developed in Node.js and will employ a MongoDB database to make data persistent.

The client will be a a single page web app made with Angular. The page will be adaptable to any screen. Also, It will have some components with tactile support, as at consequence of that, it will improve the experience on smartphones and tablets. When we use Angular, we pursuit to make our application available to desktop apps or to smartphones in the future, using frameworks like Electron or Ionic.

Palabras clave

Servicio, servidor, música, streaming, adaptativo, cliente, Node, Angular, DASH, MPEG-DASH, REST, RESTful, MongoDB, NoSQL, OAUTH2

Keywords

Service, server, music, streaming, adaptative, client, Node, Angular, DASH, MPEG-DASH, REST, RESTful, MongoDB, NoSQL, OAUTH2

Agradecimientos

Quiero dar las gracias a todas esas personas que comparten sus conocimientos en Internet y ayudan a resolver dudas y errores (como en StackOverflow). Gracias a ellos aprender cosas nuevas resulta más fácil.

ÍNDICE DE CONTENIDOS

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

1 Introducción

1.1 Motivación

Día a día Internet gana más terreno en nuestras vidas. Estamos todo el día conectados a Internet, es un poderoso canal de ventas, de noticias, de servicios, ha facilitado el enlace entre individuos a la vez que potencia los tradicionales medios de marketing y de divulgación colectiva. Si nos ponemos a pensar que aplicaciones o programas utilizamos cada día seguramente todos o la mayoría necesitan de una conexión a Internet, en mi caso, lo que más utilizo a diario es un navegador Web. El tipo de aplicaciones que más gente utiliza son las redes sociales, como Facebook o Twitter, aplicaciones de comunicación, como WhatsApp o Telegram y las de multimedia-sociales, como YouTube, Spotify o Instagram. En este trabajo empezaremos a crear una aplicación inspirada en las últimas aplicaciones nombradas.

1.2 Objetivos

El objetivo principal del proyecto es aprender a crear servicios y clientes de software que utilicen estos servicios. Para ello construiremos un servicio de streaming de música bajo demanda, es decir, capaz de servir un flujo de música concreto en el momento exacto que el usuario lo desee, y un cliente web que hará uso de este servicio. Tanto los servicios como los clientes serán diseñados y desarrollados haciendo uso de las últimas tecnologías y estándares.

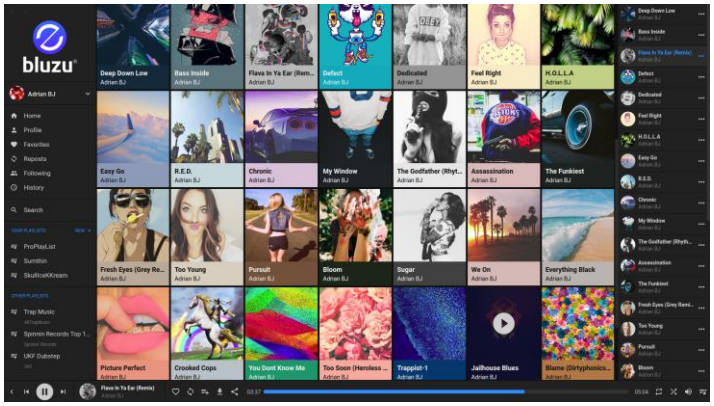


Figura 1-1: Ejemplo de la aplicación cliente que crearemos.

La mayoría de los servicios más populares utilizan una API REST para permitir a terceros hacer aplicaciones utilizando sus servicios. REST (Representational State Transfer) es un tipo de arquitectura de desarrollo web y un estándar para crear APIs para servicios orientados a Internet. Podría definirse como una interfaz entre sistemas que usan HTTP para obtener datos o generar operaciones sobre esos datos en todos los formatos posibles, como JSON o XML. También suele utilizarse el protocolo OAUTH2 [explicado en 3.2.5] para autenticar a los usuarios; con este protocolo se previene que las contraseñas de los usuarios se guarden en las aplicaciones cliente. Recientemente apareció un nuevo estándar abierto para realizar

streaming adaptativo, MPEG-DASH [explicado en 3.2.4]; varios grandes proveedores como YouTube o Netflix lo están utilizando actualmente. Uno de los objetivos de este trabajo es mostrar los pasos necesarios y el modo de utilizar la tecnología para diseñar e implementar un servicio de este tipo. El proyecto final realizado para este trabajo es totalmente adecuado como una primera versión para posteriormente construir algo mucho más grande si se desea.

1.3 Organización de la memoria

La memoria se estructura en 7 partes:

1. Introducción.
2. Estado del arte. Veremos 2 ejemplos de aplicaciones/servicios existentes.
3. Análisis y Diseño. Se explican los requisitos de la aplicación desarrollada en este trabajo, la tecnología utilizada y la arquitectura de software resultante.
4. Desarrollo. Se explica, dentro de las posibilidades dadas por las restricciones de longitud de la memoria, cómo funciona la aplicación desarrollada.
5. Integración, pruebas y resultados.
6. Conclusiones y trabajo futuro.
7. Anexos. Se detalla información útil complementaria.

2 Estado del arte

Como nuestra aplicación/servicio está centrado en el streaming de música, antes de empezar con el diseño y el desarrollo de este proyecto, veremos 2 de las empresas más exitosas en este ámbito: Spotify [20] y SoundCloud [23]. Ambas empezaron con desarrollos relativamente pequeños y han ido creciendo hasta lo que son hoy en día. Existen más empresas que pueden ser válidas para ver en este punto, todas ofrecen más o menos lo mismo, aunque cada una tiene sus detalles que pueden hacer que más gente se sienta atraída por su servicio y no por otro.

Las dos empresas a continuación han sido elegidas sobre otras por dos razones. La primera es que personalmente las utilizo y la segunda es que tienen una documentación extensa y un blog en el que van contando sus aventuras desarrollando sus servicios que me han inspirado para realizar este trabajo.

2.1 Spotify

Spotify ofrece un servicio de streaming de música gratuito con anuncios y otro de pago sin anuncios. Para subir música a este servicio debes de ser un artista: cualquiera no puede empezar a subir canciones. Su API tampoco permite reproducir canciones de más de 30 segundos excepto que el usuario que utiliza la aplicación tenga una cuenta premium. Esto lo hacen porque se preocupan mucho del copyright e intentan que no se puedan descargar las canciones fácilmente. No hay demasiada información sobre la arquitectura interna del

servicio. Se están centrando mucho en Inteligencia Artificial para intentar crear las mejores recomendaciones posibles de canciones adaptadas a cada usuario, incluso han adquirido una startup, Niland (niland.io), dedica al analisis de audio mediante el uso de Inteligencia Artificial.

2.2 SoundCloud

SoundCloud se parece un poco a un Twitter de música: es completamente gratuito y sin anuncios. En SoundCloud no pagan los que escuchan, sino los que quieren subir canciones, podcast, mixes, etc. En su blog [24] cuentan cómo empezaron con una arquitectura monolítica utilizando el framework de Rails. No era la mejor forma de implementar el servicio como el que quieren ofrecer, si la idea es escalar a varios millones de usuarios. Por ese motivo han ido migrando esa arquitectura monolítica a otra basada en microservicios, donde cada microservicio se encarga de una tarea muy específica. También ofrecen recomendaciones de canciones, pero no son tan avanzadas como las de Spotify. Su recomendación está basada en recomendación por ítem principalmente.

Estas empresas al igual que otras, tiene solucionado el tema de la arquitectura del servicio, aunque siempre están mejorando. Ahora lo que se podría decir que está de moda es la Inteligencia Artificial, e intentan constantemente mejorar sus servicios de recomendación.

3 Diseño

3.1 Requisitos

3.1.1 No funcionales

1. **Fácil de usar por el usuario.** Creando un diseño simple, con el menor número de componentes visibles en cada vista, y una apariencia un poco familiar.
2. **Cliente multiplataforma**, sin necesidad de crear una aplicación nativa para cada dispositivo. Crear una aplicación nativa para cada dispositivo lleva mucho tiempo, costes y esfuerzo.
3. **Fácil de programar y mantener.** Utilizando un diseño modular y un lenguaje fácil de usar, entender y mantener.
4. El **cliente** podrá ser desarrollado en **cualquier plataforma**, para el **servidor** se usará **Linux**, pues ofrece mejor rendimiento y muchas herramientas, como un compilador de C/C++, vienen por defecto incluidas.
5. **Utilizar las últimas tecnologías y estándares.** Estas tecnologías y estándares son usadas y mantenidas por empresas importantes de gran tamaño, además de ser de código libre. Además, podremos prepararnos para el futuro.
6. **Rápido para unos pocos miles de usuarios.** La aplicación (servidor + cliente) por ahora formará la base para algo mayor. Solo será necesario contar con un servidor.
7. **Con posibilidades de hacerlo escalable en el futuro.** Si se decide continuar con el proyecto, debe ser fácil adaptarlo sin necesidad de hacer grandes cambios.
8. **Seguridad.** Es importante proteger los datos personales de los usuarios así como sus datos obtenidos tras el uso de la aplicación. Para ello los clientes tendrán que contar con el permiso del usuario para acceder a sus datos.
9. **Mantener la calidad del audio y poder reproducirlo sin cortes**, o con el menor número de cortes posibles. Esto dependerá también de la calidad de conexión del usuario.
10. Para **facilitar el desarrollo inicial** solo se podrá subir audio en formato mp3 e imágenes en formato jpg.

3.1.2 Funcionales

1. Login y registro de usuarios / Borrar cuenta.
2. Crear/eliminar/modificar canciones y añadir audio.
3. Crear/eliminar/modificar listas de reproducción.
4. Subir/añadir imágenes de portada a las canciones y las listas de reproducción.
5. Reproducir canciones.
6. Cola de reproducción.
7. Mantener la ‘sesión’ al salir de la aplicación, mientras que no se haga logout.
8. Permitir registrar clientes a usuarios logueados.
9. Solicitar acceso al usuario para permitir a una aplicación cliente acceder a sus datos.

3.2 Tecnologías a utilizar

A continuación, mostraré las principales tecnologías usadas en este proyecto: entornos, frameworks, protocolos, etc. El resto de bibliotecas y herramientas menos centrales al desarrollo serán nombradas según sea necesario, explicando brevemente en qué consisten.

3.2.1 Node.js

Node.js [1] es un entorno de ejecución basado en JavaScript, orientado a eventos asíncronos y con operaciones de entrada y salida sin bloqueo, esto lo hace liviano y eficiente. Utiliza el motor de JavaScript V8 de Google, que está escrito en C++ y compila el código JavaScript a código máquina en lugar de interpretarlo en tiempo real; este motor es utilizado también en el navegador Chromium/Chrome. Fue creado con la idea de ser útil en la creación de programas de red altamente escalables y para situaciones con una gran cantidad de tráfico, donde la lógica del lado del servidor y el procesamiento requeridos, no sean grandes antes de responder al cliente.

Las operaciones de red basadas en hilos son relativamente ineficientes y difíciles de usar. Además, el proceso nunca se bloquea porque casi ninguna función de Node realiza I/O directamente. HTTP es ciudadano de primera clase en Node, diseñado con operaciones de streaming y baja latencia.

Es capaz de manejar muchas conexiones concurrentes. Funciona con un modelo de evaluación de un único hilo de ejecución, usando entradas y salidas asíncronas las cuales pueden ejecutarse concurrentemente en un número de hasta cientos de miles sin incurrir en costos asociados al cambio de contexto. Aunque esté diseñado con un solo hilo, se pueden utilizar todos los cores del procesador. Se pueden crear procesos hijos haciendo uso de fork como en otros lenguajes, también está el módulo clúster, que permite compartir sockets entre procesos para activar el balanceo de carga en sus múltiples cores.

En Java o PHP cada conexión genera un nuevo hilo que potencialmente viene acompañado de 2 MB de memoria, con 8GB de RAM se podrían tener hasta un máximo teórico de 4000 conexiones. En Node cada conexión genera un evento dentro del proceso del motor de Node y se ejecuta un callback, cada conexión recibe una pequeña asignación de espacio de memoria dinámica sin tener que generar un hilo de trabajo. Node afirma que puede tener decenas de miles de conexiones concurrentes.

Presenta un bucle de eventos como un entorno y no una librería, en otros sistemas siempre existe una llamada que bloquea para iniciar el bloque de eventos. Está pensado para utilizar como servidor, pero también se puede utilizar para crear aplicaciones de escritorio, por ejemplo.

3.2.2 Redis

Redis es una estructura de datos en memoria usada como base de datos, caché y broker de mensajes. Soporta múltiples tipos de estructuras de datos como strings, listas, conjuntos, etc.

y operaciones útiles como concatenación, incrementación de un valor hash, unión e intersección de conjuntos, etc. También puede utilizar almacenamiento en disco parcial o completo. Soporta la replicación del tipo maestro-esclavo pudiéndose replicar los datos de un servidor a muchos esclavos. En escenarios de datos no durables (solo usando memoria RAM) el rendimiento puede ser extremo comparado con motores de bases de datos, tampoco hay una notable diferencia entre lectura y escritura de datos.

Recomiendan Linux para el despliegue. No soportan Windows directamente, pero existe una portación.

3.2.3 MongoDB

MongoDB es un sistema de bases de datos NoSQL orientado a documentos. Guarda los datos en estructuras de datos de tipo JSON, en concreto BSON, en vez de utilizar tablas como en SQL. BSON es una representación binaria de estructuras de datos y mapas.

Soporta búsqueda por campos, consultas de rangos y expresiones regulares. Cualquier campo de un documento de MongoDB puede ser indexado. El concepto de índice en MongoDB es similar a los encontrados en bases de datos relacionales.

Fue creada para dar escalabilidad, rendimiento y gran disponibilidad, escalando una implementación de único servidor a grandes arquitecturas complejas de centros multidatos. Soporta replicación de tipo primario-secundario y sharding, lo que la hace que sea una fácilmente escalable. El sharding es como tener una base de datos dividida en varias bases de datos. MongoDB hace transparente el acceso a estos datos distribuidos.

Una de las diferencias más importantes respecto a las bases de datos relacionales es que no es necesario seguir un esquema. Los documentos de una misma colección (tabla en base de datos relacional) pueden tener esquemas diferentes.

El principal problema de esta base de datos es que no implementa las propiedades ACID (Atomicidad, Consistencia, Aislamiento y Durabilidad) necesarias para realizar transacciones.

MongoDB bloquea la base de datos a nivel de documento ante cada operación de escritura. Solo se podrán hacer escrituras concurrentes entre distintos documentos.

3.2.4 MPEG-DASH

Actualmente las tecnologías de streaming adaptativo sobre HTTP son: HLS (Apple HTTP Live Streaming), HDS (Adobe HTTP Dynamic Streaming) y MSS (Microsoft Smooth Streaming), todas propietarias, y MPEG-DASH (MPEG Dynamic Adaptive Streaming over HTTP), el único internacional y estandarizado. Actualmente está siendo utilizado por empresas importantes como YouTube y Netflix.

La idea básica es generar múltiples versiones del mismo contenido, cada uno con diferentes calidades, cambiando el bitrate por ejemplo, y dividir estas diferentes versiones en segmentos de, por ejemplo, 2 segundos cada uno. Los segmentos se encuentran en un servidor Web y se pueden descargar enviando una petición GET sobre HTTP. Normalmente la relación entre los distintos archivos está descrita en un archivo de manifiesto (un ejemplo de este archivo se encuentra en el Anexo E). Para hacer streaming es necesario que el cliente coja este archivo para conocer qué opciones tiene, los tiempos de los segmentos, además de que cada segmento se puede encontrar en un servidor distinto.

La adaptación al bitrate o calidad del archivo la realiza el cliente, dependiendo de la velocidad de la red; puede descargarse segmentos de distintas calidades intercalados y reproducirlos como una sola pista.

MPEG-DASH, al igual que Apple HLS, pueden ser usados con servidores HTTP comunes como Apache, NGINX, etc. Adobe y Microsoft necesitan más lógica/mecanismos en el servidor.

El protocolo ofrece bastantes características como: múltiples canales audio, se puede cambiar el idioma en cualquier momento; protección de contenido flexible con encriptación común (DRM), subtítulos, inserción de publicidad eficiente, cambio de canal rápido, múltiples CDN en paralelo, soporta HTML5, es agnóstico a los códecs de audio, soporta el contenido no multiplexado (audio y video por separado), define métricas de calidad, es tolerante a fallos en el cliente, es capaz de añadir o quitar niveles de calidad durante el streaming y alguna característica más.

El principal problema que tiene actualmente es que es bastante joven y no todos los reproductores tienen soporte o tienen implementadas todas sus características.

3.2.4.1 FFMPEG

Es una colección de software libre que puede grabar, convertir y hacer streaming de audio y video. Aunque está desarrollado en Linux puede ser compilado para el resto de sistemas operativos. Lo utilizaremos para convertir los archivos de audio en mp3 al formato m4a y para crear distintas versiones con distintos bitrates.

3.2.4.2 MP4Box

Es un empaquetador de multimedia que se encuentra dentro del framework multimedia de código libre GPAC. Utilizaremos este empaquetador para crear los segmentos para el streaming adaptativo MPEG-DASH. El ejemplo del Anexo E ha sido generado utilizando esta herramienta.

3.2.5 OAUTH 2

Es el protocolo estándar de autorización para la industria. Permite a terceros (clientes) acceder a contenidos propiedad de un usuario (alojados en aplicaciones de confianza, servidor de recursos) sin que éstos tengan que manejar ni conocer las credenciales del usuario. Es decir, aplicaciones de terceros pueden acceder a contenidos propiedad del usuario, pero estas aplicaciones no conocen las credenciales de autenticación.

Hay 4 flujos de autorización, cada uno indicado para una cosa, que son: código de autorización, implícito, credenciales del propietario del recurso y credenciales del cliente. Solo explicaré el primer flujo, que es el que vamos a utilizar en la aplicación.

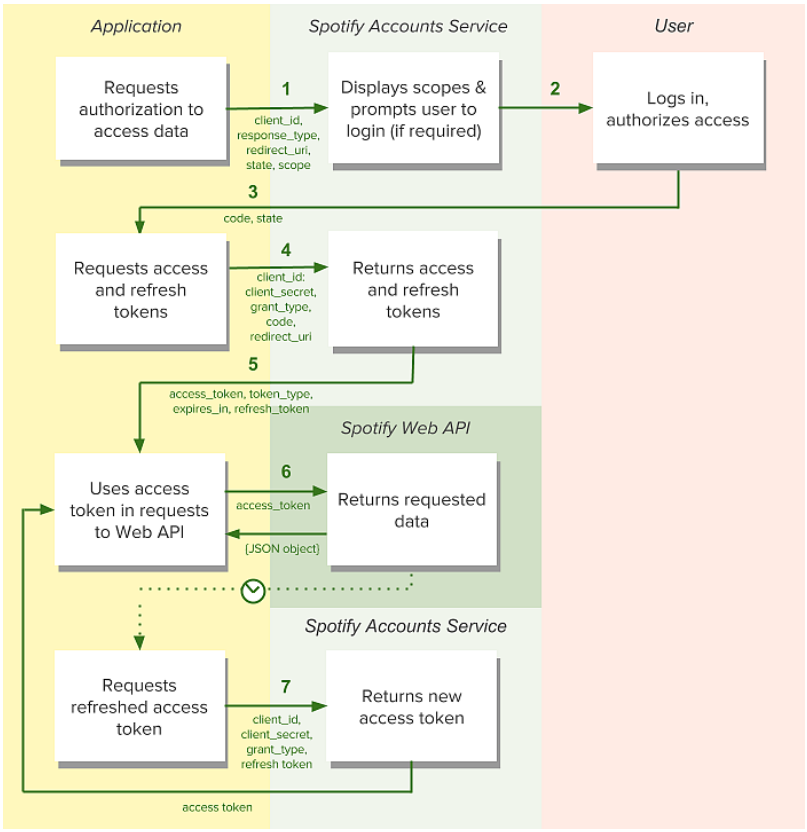


Figura 3-1: Flujo de autorización de OAUTH2 por código de autorización. Fuente: Spotify [14].

El flujo por código de autorización comienza cuando el usuario quiere hacer login en un cliente. (1) El cliente envía una petición al servidor de cuentas con varios datos necesarios para identificar al cliente como son el ID y la URI de redirección. (2) El servidor de cuentas muestra una ventana de login al usuario, independiente de la aplicación cliente. (3) El usuario mete su nombre de usuario y contraseña y da permiso a la aplicación a acceder a sus datos. Se le envía un código a la aplicación cliente. (4) Con el código recibido, el ID, la clave secreta y la URI de redirección del cliente, se pide el token de autorización. (5) Se envía el token al cliente, junto con un tiempo de expiración y un token de refresco o renovación que permite volver a pedir un token nuevo cuando expire el recibido sin necesidad de que el usuario vuelva a hacer login. (6) Se utiliza el token para acceder a los datos del usuario. (7) Se pide un nuevo token utilizando el token de refresco.

3.2.6 Angular

Angular es un framework web creado por Google para crear aplicaciones web de una sola página. Es una evolución de AngularJS, aunque es incompatible con él. Se puede utilizar para crear aplicaciones para distintas plataformas, web, móvil web, móvil nativo y escritorio, utilizando otros frameworks como Ionic, Native Script o Electron.

El lenguaje principal elegido por Google para programar en Angular es TypeScript, también se puede utilizar JavaScript directamente o Dart. TypeScript permite crear un código más claro y organizado además de que al utilizar un compilador nos ayuda a resolver bastantes problemas antes de ejecutar nuestra aplicación. TypeScript puede ser compilado a cualquier versión de JavaScript.

En Angular se utilizan componentes que siguen un modelo parecido al Modelo Vista Controlador. Un componente está formado por una combinación de una plantilla HTML (más otra CSS) y una clase que se encarga de controlar la porción de pantalla que ocupa el componente. Cada componente empieza por una función decorador @Componente que tiene como parámetro un objeto con los metadatos del componente. Cada componente es independiente, están encapsulados para que no se afecten por el DOM general excepto que se lo permitamos nosotros. Aparte de los componentes, también existen los servicios, directivas y módulos. Los servicios son realmente útiles para realizar tareas concretas que utilizan uno o varios componentes. También se pueden utilizar para comunicar componentes. Las directivas se utilizan en las plantillas HTML y sirven para modificar lo que muestra el componente y para obtener datos de su DOM. Los módulos sirven para agrupar componentes.

3.3 Servidor

3.3.1 Visión general

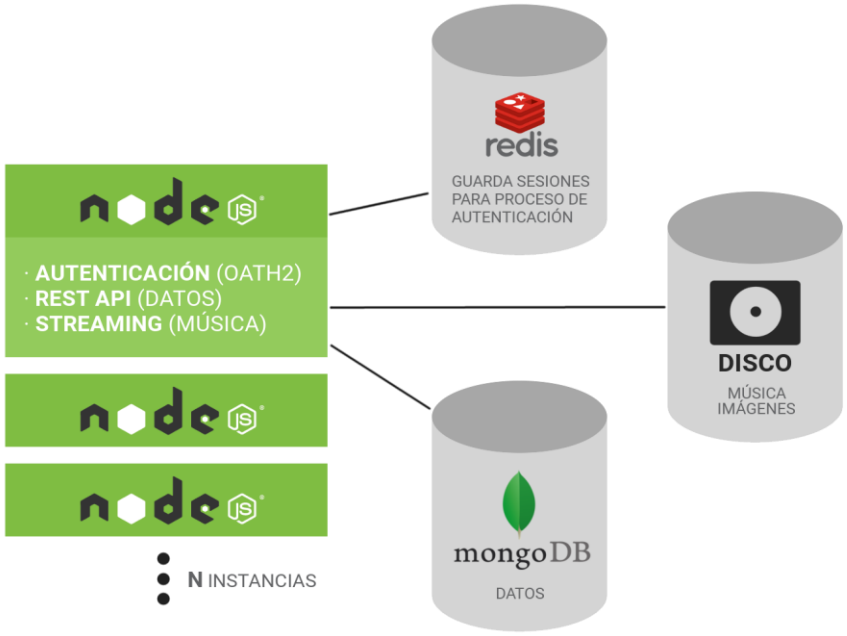


Figura 3-2: Diseño general del servidor

Ahora que ya tenemos una idea de que tecnologías vamos a usar y en qué consisten, vamos a continuar explicando el diseño del servidor. Nuestro servidor constará de una aplicación escrita en Node.js de la que puede haber más de una instancia. La razón de utilizar más de una instancia es la de aprovechar todos los procesadores disponibles del computador en el que se encuentre, pues Node.js solo funciona en un único hilo y todas las peticiones en principio se procesan sólo con ese hilo de ejecución.

La aplicación en Node se encargará de todo: tanto de la autenticación de los usuarios y los clientes, como de la API REST para intercambio de datos, como del streaming, el procesamiento de los archivos, entre otras funciones. También dispondremos de dos bases de datos, una Redis y otra MongoDB. Utilizaremos Redis indirectamente, pues ya se encargará una librería de utilizarla por nosotros. Esta librería se usará para el proceso de autenticación de los usuarios, pues es necesario guardar temporalmente una sesión por cada usuario para identificarlo a él y al cliente que está usando mientras se completa la autenticación siguiendo el protocolo OAUTH2. Por otro lado, MongoDB lo utilizaremos para guardar los datos de las canciones, listas de reproducción, usuarios, clientes, además de también utilizarla para la autenticación, porque guardaremos los tokens obtenidos por el proceso de autenticación de OAUTH2, para que cuando recibamos peticiones a la API podamos comprobar si es válido el token de cada petición.

Finalmente tenemos el disco, aquí guardaremos las canciones y las imágenes. La decisión de guardar todas las imágenes y canciones en el mismo servidor en el que se encuentra la

aplicación de Node no nos permitirá crear un clúster, o varios, de servidores con la aplicación Node, aunque para este trabajo tampoco nos es necesario del todo. La forma de resolver este problema sería creando un servicio de almacenamiento independiente del resto, hablaré un poco más de este tema en el punto 6.2 (Trabajo futuro).

3.3.2 Autenticación/Autorización

Como ya se dijo, para la autenticación de los usuarios y los clientes la haremos haciendo uso del protocolo OAUTH2. Solo implementaremos una forma de autenticación, recordamos que hay varias formas de autenticarse y que también se puede dar acceso a más o menos recursos/datos del usuario o durar por más o menos tiempo el tiempo válido del token; los detalles ya se dieron en el punto 3.2.5.

El flujo de autenticación que utilizaremos será el del código de autorización, pero no proporcionaremos un token de actualización para permitir pedir un nuevo token de acceso sin volver a realizar el proceso de login por parte del usuario. También permitiremos acceso total a la información del usuario, sin dar a elegir al usuario si solo quiere permitir que la aplicación solo vea sus listas de reproducción, las canciones que ha subido o a qué canciones le han gustado, por poner un ejemplo. La forma ideal para aplicaciones que se ejecutan en el navegador o en cualquier otra aplicación de usuario, sería utilizar solo el flujo de autorización implícita, y si la aplicación se ejecuta en un servidor se utiliza el flujo del código de autorización, pues desde las aplicaciones accesibles por los usuarios se puede llegar a ver el token secreto, pero no en el caso de que se encuentre en un servidor. Aun así seguiremos con esta implementación, pues se podría decir que es la más compleja, y entender esta hará más fácil implementar los otros flujos en un futuro.

El flujo de autenticación comienza cuando un usuario pincha el botón de login de la aplicación cliente. Al pinchar el botón la aplicación debería de redirigir a la siguiente url:

http://192.168.1.111:8080/oauth2/authorize?client_id=59209785ef9cc509f97b42e3**&response_type=**code**&redirect_uri=**http://192.168.1.111/

Para esta URL el navegador hará una petición GET y el servidor le responderá con una página en la que el usuario deberá hacer login o registrarse haciendo uso de los formularios que aparecen**. 192.168.1.111:8080** es la IP del host y el puerto en el que se está ejecutando el servidor de Node. **/oauth2/authorize** es el endpoint en el que se comienza el flujo de autenticación. Todo lo que se encuentra después de la interrogación son parámetros. Los parámetros necesarios para este endpoint son: **client_id**, es el id del cliente; **response_type**, es el tipo de respuesta que se quiere, en este caso **code** (código); y **redirect_uri**, que indica la url a la que se debe redirigir en caso de que los parámetros sean correctos, el usuario haga login ó se registre y además de permiso a la aplicación a acceder a los datos. En este caso la URL de **redirect_uri** tiene la misma IP que el servidor de Node porque la aplicación web cliente es servida por este host también. Para que el client_id y redirect_uri sean correctos, previamente se ha tenido que registrar el cliente en el servidor, especificando la URL de redirección.

A continuación, en la figura 3-2, se muestran los formularios de login y registro. Los dos formularios se encuentran en la misma página, los dos botones de arriba, de color azul, son botones de navegación. Estos botones mostrarán u ocultaran un formulario u otro.

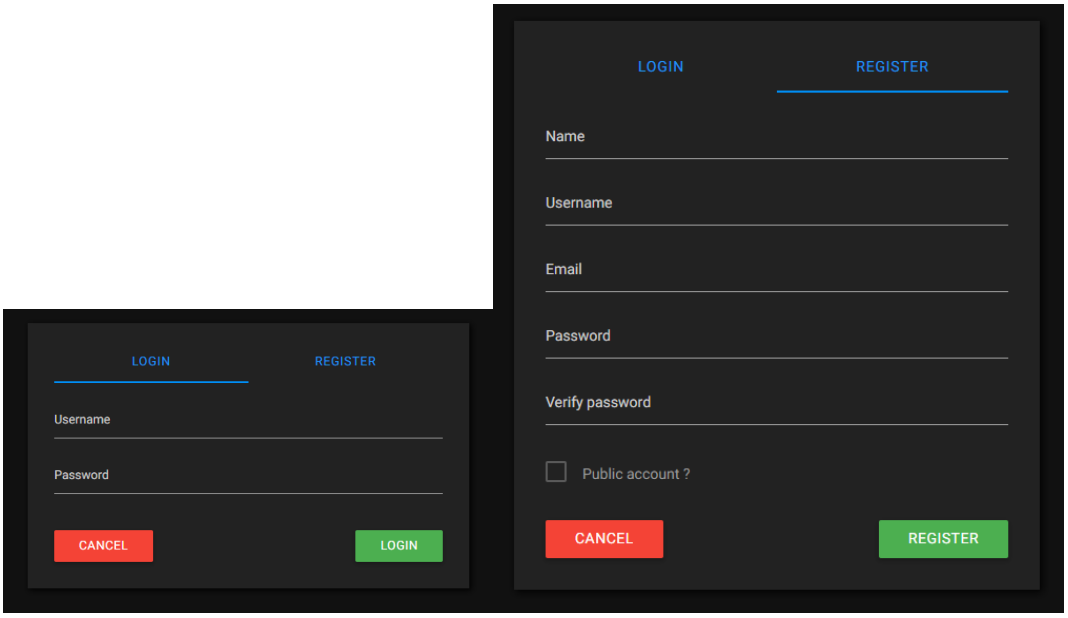


Figura 3-3: Página con formularios de login y registro.

Al pulsar en algún botón **cancel**, se volverá a la página web del cliente. Al pulsar el botón de **login** o **register**, se hará una petición a la misma URL que arriba (**http://192.168.1.111:8080/oauth2/authorize?client_id=**59209785ef9cc509f97b42e3**&response_type=**code**&redirect_uri=**http://192.168.1.111/), pero esta vez se utilizara POST en vez de GET. Al realizar el POST al servidor también enviamos los datos del formulario correspondiente dentro de la petición. Si los datos introducidos por el usuario no son válidos se mostrará un error indicandolo, en caso de que, si sean válidos, el servidor responderá con otra página con otro formulario, pasando así al siguiente paso del flujo.

El siguiente formulario se muestra en la figura 3-3. Podemos ver que en color azul se resalta el nombre del usuario que ha hecho login o se acaba de registrar, el nombre de la aplicación cliente y finalmente a que tendrá acceso la aplicación, en nuestro caso será a todo siempre. Esta página web también tiene oculto un id de transacción, que se enviará con el formulario al pulsar uno de los dos botones. Este id se usa para identificar a qué usuario y cliente pertenece la próxima petición.

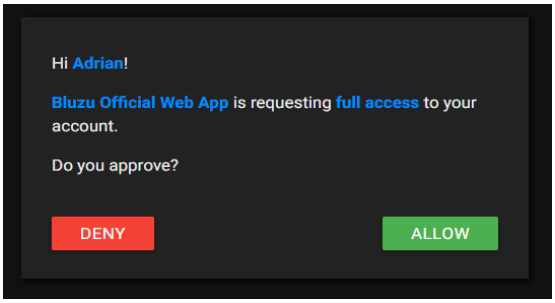


Figura 3-4: Página con formulario para dar permiso a la aplicación a acceder a los datos.

Al pulsar uno de los dos botones se enviará una petición POST a la siguiente URL:

http://192.168.1.111:8080/oauth2/decision

Si el usuario pulsa el botón **deny** se parará el flujo y se redirigirá a la URL del cliente. Si por el contrario, pulsa el botón **allow**, el servidor enviará al navegador una petición de redirección con la URL indicada en **redirect_uri** más el código que se estaba pidiendo como parámetro:

http://192.168.1.111/?code=ev8bo5xmout1s5f8

Este código tendrá que usarlo la aplicación cliente para realizar el último paso de la autenticación/autorización. El último paso es el de pedir el token de autorización. Cuando la aplicación cliente reciba esa URL, cogerá el código y hará una petición POST a la siguiente URL:

http://192.168.1.111:8080/oauth2/token?code=ev8bo5xmout1s5f8**&grant_type=authorization_code&redirect_uri=**http://192.168.1.111/

Los parámetros usados son: **code**, es el código que acabamos de recibir del servidor, **grant_type**, es el tipo de flujo que estamos utilizando, en nuestro caso, **authorization_code** (código de autorización), y de nuevo, **redirect_uri**, con la url de redirección. Como todos los parámetros son correctos, el servidor responde con un JSON que contendrá el token.

3.3.3 Modelos/Esquemas BBDD

Con las bases de datos NoSQL podemos guardar en las colecciones lo que queramos, no importa la estructura que tengan, y esto puede tener ventajas como desventajas. La desventaja en nuestro caso puede ser esa misma característica, el poder guardar cualquier cosa, pues si queremos que nuestros documentos tengan una estructura que cumplan todos los documentos introducidos en una colección, para de esta forma intentar que sea más mantenible y entendible, necesitamos la ayuda de otra herramienta. La herramienta elegida es Mongoose, que nos ayuda a modelar nuestros datos de una forma fácil mediante la utilización de esquemas, además de otras ventajas como conexión a la base de datos, validación, casting o consultas, y está desarrollada para Node.js. Si intentamos introducir un documento en una colección que no cumple el esquema, Mongoose no lo introduce.

A continuación, en la figura 3-4, podemos observar las relaciones que hay entre los esquemas que vamos a utilizar:

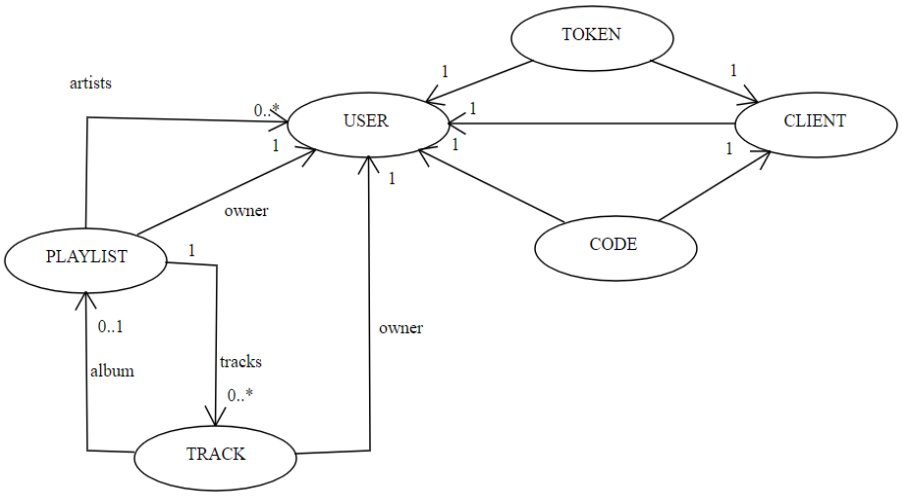


Figura 3-5: Relaciones entre esquemas

Para ayudar a entender el grafo anterior vamos a ver cómo definimos los esquemas. Por defecto todos los documentos tendrán un atributo '_id' que no se puede quitar, aunque definamos otros índices, y que hará de clave primaria, si pensamos como en SQL. Si nosotros no proporcionamos uno, MongoDB creará uno. También podemos crear índices utilizando otros atributos, como por ejemplo el nombre de usuario en el caso de los usuarios. Para relacionar unos documentos con otros utilizaremos este _id. Para obtener documentos la mejor forma es usando también este índice, pues, aunque tengamos varios millones de documentos la recuperación del documento tarda unos pocos milisegundos. En los siguientes esquemas aparecerá comentado, para recordar que existe en el documento, pero no es necesario definirlo en el esquema.

El esquema de un usuario sólo guarda información del usuario, no tiene ninguna referencia a otro documento de otra colección. La información que guardaremos será el nombre de usuario, el email, URL de la imagen de perfil, URL de la imagen de fondo del perfil, el nombre, contraseña, si la cuenta es pública, país y descripción.

Es importante destacar que Mongoose permite personalizar los documentos que se devuelven a los clientes. Gracias a esto podemos definir que no queremos devolver nunca la contraseña, por ejemplo. También podemos añadir más atributos a cada documento de la respuesta, por ejemplo, he añadido el atributo 'type', que puede tener como valores 'user', 'playlist' o 'track' para hacer más fácil saber qué tipo de documento es. Estos atributos se añaden a los documentos que se envían como respuesta al cliente, no se guardan en los documentos en la base de datos, pues no es necesario.

Podemos observar que se puede exigir que haya un número mínimo de atributos en el documento definiéndolos como 'required'. También, para asegurarnos que un atributo sea único en toda la colección utilizaremos 'unique'. 'require' y 'unique' tienen por defecto el valor 'false'. También se puede utilizar un valor por defecto haciendo uso de 'default'. Todos los usuarios harán uso de las mismas imágenes por defecto hasta que suban otras imágenes que ellos quieran.

```
let UserSchema: any = new Schema({
    // _id          : tipo ObjectId, por defecto puesto siempre por Mongo
    username        : { type: String, unique: true, required: true },
    email           : { type: String, required: true },
    profileImage    : { type: String, default: "images/default/no-profile.jpg" },
    backgroundImage : { type: String, default: "images/default/no-backgrnd.jpg" },
    name            : { type: String, required: true },
    password        : { type: String, required: true },
    public          : { type: Boolean, default: true },
    country         : String,
    description     : String,
});
```

Figura 3-6: Esquema para los usuarios.

El siguiente esquema es para las listas de reproducción. La información que se guardara sobre ellas es, la fecha de creación, que se añadirá automáticamente al crear el documento, una descripción, géneros, una imagen con más atributos que ahora veremos, una etiqueta que indica si es un álbum, discográficas, para el caso de que sea un álbum, propietario, es el usuario que la ha creado, fecha de salida, también para el caso de que sea un álbum, y finalmente una lista de canciones.

Entrando un poco más en detalle, el atributo imagen guarda un objeto que tiene varios atributos que son, el color dominante, el color de texto que mejor se ve con el color dominante, la URL con la que se puede coger la imagen, una lista con las calidades/resoluciones de la imagen y el formato, que será jpg, jpeg o png. El formato jpg y jpeg son el mismo, pero si la imagen se guarda con extensión jpg y pones jpeg, el servidor no la va a encontrar.

El directorio de imágenes y el de audio los he definido en la aplicación de Node como directorios estáticos, y haciendo una llamada GET a la URL http://<ip>/images/<nombre_imagen> y http://<ip>/audio/<nombre_audio> se puede acceder a los archivos que se encuentran en los directorios en los que se guardan las imágenes y el audio. Para acceder al audio veremos una forma mejor forma de hacerlo más adelante. Para las imágenes, para no complicarnos accederemos a ellas de esta forma.

En un primer momento hice un esquema para álbumes también, pero era muy parecido a las listas de reproducción y decidí unirlos. Esa es la razón por la que las listas de reproducción tienen atributos, como ‘isAlbum’, ‘labels’, o ‘releaseDate’.

Por último, comentaremos la lista de canciones. Esta lista está formada por objetos con 4 atributos que son, _id, fecha en la que se añadió, quién la añadió y el id de la canción. Ahora mismo puedes estar preguntándote porque aparece ‘_id’ aquí también. La explicación es sencilla, cuando se crea una lista de objetos, se crea una subcolección. Solo se crea como subcolección cuando son objetos, una lista de String ó Number es una lista como las de siempre.

```
let PlaylistSchema = new Schema({
   // _id       : tipo ObjectId, por defecto puesto siempre por Mongo
   createdDate  : { type: Date, default: Date.now },
   description  : String,
   genres       : [String],
   image        : { type: {
                        dominantColor : String,
                        textColor     : String,
                        url           : String,
                        qualities     : [Number],
                        format        : String.
                    }, default: {dominantColor: "#323232", textColor:"#eee", url:
"images/default/no-cover.jpg", format: "jpg" }},
   isAlbum      : { type: Boolean, default: false },
   labels       : [String],
   name         : { type: String, required: true },
   public       : { type: Boolean, default: false },
   owner        : { type: Schema.Types.ObjectId, ref: 'User', required: true },
   releaseDate  : Date,
   tracks       : { type: [{
                        // _id
                        addedAt : { type: Date, default: Date.now },
                        addedBy : { type: Schema.Types.ObjectId, ref: 'User',
required: true },
I                       trackId : { type: Schema.Types.ObjectId, ref: 'Track',
required: true }
                    }]}
});
```

Figura 3-7: Esquema para las listas de reproducción.

El siguiente esquema pertenece a las canciones. Tiene una relación opcional a una lista de reproducción que debería de estar marcada como album. También tenemos dos listas para los artistas, la idea de crear dos listas es guardar en la de 'artists' artistas que estuvieran registrados en la plataforma, y en la de 'artistsN' guardar el nombre de los artistas que no estén registrados. De esta forma, al estar el usuario registrado se puede poner un link hacia su perfil. Otros atributos de las canciones son, la duración, guardada en segundos, si tiene letras explicitas, los géneros, si es posible reproducirla (ahora explicare esto), una imagen, igual que las listas de reproducción, nombre de la canción, quien ha subido la canción, si es pública, la fecha de salida, las URL para realizar streaming, y finalmente el número de pista, si es que pertenece a un álbum.

El atributo 'isPlayable', que indica si se puede reproducir la canción o no, se debe a que antes de poder subir el audio o una imagen, es necesario haber creado en la base de datos un documento con los atributos mínimos, que en este caso solo es el _id. Veremos más en detalle esto en la parte del cliente, pero básicamente al hacerlo así, me ha permitido hacer en el cliente que he hecho, poder subir el audio, la imagen y actualizar los datos, como nombre, artistas, etc. a la vez y de forma independiente, en vez de hacer un formulario, en el que se rellenen todos los campos y se indique la canción y la imagen que se quieren subir, porque no empezarán a subirse al servidor hasta que se pulse el botón de enviar formulario.

```
let TrackSchema = new Schema({
    // _id      : tipo ObjectId, por defecto puesto siempre por Mongo
   albumId     : { type: Schema.Types.ObjectId, ref: 'Playlist' }, // Id to playlist that
is an album
   artists     : { type: [Schema.Types.ObjectId], ref: 'User' }, // artists with an
account
   artistsN    : [String], // artists without an account
   createdDate : { type: Date, default: Date.now },
   duration    : { type: Number, default: 0 }, // in seconds
   explicit    : { type: Boolean, default: false },
   genres      : [String],
   isPlayable  : { type: Boolean, default: false },
   image        : { type: {
                      dominantColor : String,
                      textColor     : String,
                      url           : String,
                      qualities     : [Number],
                      format        : String
                  }, default: {dominantColor: "#323232", textColor:"#eee", url:
"images/default/no-cover.jpg", format: "jpg" }},
   name        : { type: String },
   owner       : { type: Schema.Types.ObjectId, ref: 'User', required: true },
   public      : { type: Boolean, default: false },
   releaseDate : Date,
   url         : { type: {
                      html5 : String,
                      dash  : String
                  }},
   trackNumber : Number,
});
```

Figura 3-8: Esquema para las canciones.

A continuación, tenemos el esquema que define a los clientes. La información que guardamos de los clientes, es, un nombre, una clave secreta, el id del usuario que la ha creado y la URL de redirección para el caso de que la autenticación sea válida.

```
let ClientSchema = new Schema({
   // _id      : tipo ObjectId, por defecto puesto siempre por Mongo
   name        : { type: String, required: true, unique: true },
   secret      : { type: String, required: true },
   userId      : { type: Schema.Types.ObjectId, ref: 'User', required: true },
   redirectUri : { type: String, required: true },
});
```

Figura 3-9: Esquema para definir un cliente.

El esquema de códigos, usado en el proceso de autenticación, guardará un valor, que contiene el código, la URL de redirección también, una referencia al cliente que está haciendo login y una referencia al cliente que está utilizando para hacer login.

```
let CodeSchema = new Schema({
   // _id      : tipo ObjectId, por defecto puesto siempre por Mongo
   value       : { type: String, required: true },
   redirectUri : { type: String, required: true },
   userId      : { type: Schema.Types.ObjectId, ref: 'User', required: true },
   clientId    : { type: Schema.Types.ObjectId, ref: 'Client', required: true }
});
```

Figura 3-10: Esquema para guardar el código que se genera en el proceso de autenticación.

Por último, tenemos el esquema para guardar tokens de autorización. Guardaremos una referencia al usuario y otra al cliente al que pertenece el token, y un valor, que contiene el token.

```
let TokenSchema = new Schema({
   // _id      : tipo ObjectId, por defecto puesto siempre por Mongo
   userId      : { type: Schema.Types.ObjectId, ref: 'User', required: true },
   clientId    : { type: Schema.Types.ObjectId, ref: 'Client', required: true },
   value       : { type: String, required: true }
});
```

Figura 3-11: Esquema para tokens de autenticación.

3.3.4 API REST

ACCIÓN	ENDPOINT	OAUTH2
GET	**/all/playlists**	**NO**
	Este endpoint solo sirve para el desarrollo, con el podemos obtener una lista con todas las listas de reproducción creadas.	
GET	**/all/users**	**NO**
	Este endpoint solo sirve para el desarrollo, con el podemos obtener una lista de todas los usuarios registrados.	
GET	**/all/tracks**	**NO**
	Este endpoint solo sirve para el desarrollo, con el podemos obtener una lista con todas las canciones creadas.	
GET	**/clients**	**SI**
	Devuelve una lista con los clientes creados por el usuario autenticado.	
POST	**/clients**	**SI**
	Crea un nuevo cliente para el usuario autenticado. Se le pasara los datos necesarios (nombre, URL de redirección) por el cuerpo de la petición en formato JSON.	
GET	**/oauth2/authorize**	**-**
	Usado para el proceso de autenticación. Explicado en el punto 3.3.2.	

POST	**/oauth2/authorize**	**-**
	Usado para el proceso de autenticación. Explicado en el punto 3.3.2.	
POST	**/oauth2/decision**	**-**
	Usado para el proceso de autenticación. Explicado en el punto 3.3.2.	
POST	**/oauth2/token**	**-**
	Usado para el proceso de autenticación. Explicado en el punto 3.3.2.	
GET	**/playlists?ids=<lista_ids>**	**NO**
	Devuelve una lista de listas reproducción con los IDs pasados por el parámetro ids. Los IDs van separados por comas.	
GET	**/playlists/:id**	**NO**
	Devuelve la lista con el ID pasado por parámetro.	
GET	**/playlists/:id/tracks**	**NO**
	Devuelve la lista de canciones de una lista de reproducción con ID pasado por parámetro.	
GET	**/tracks?ids=<lista_ids>**	**NO**
	Devuelve una lista de canciones con los IDs pasados por el parámetro ids. Los IDs van separados por comas.	
GET	**/tracks/:id**	**NO**
	Devuelve una canción con el ID pasado por parámetro.	
GET	**/users?ids=<lista_ids>**	**NO**
	Devuelve una lista de usuarios con los IDs pasados por el parámetro ids. Los IDs van separados por comas.	
GET	**/users/:id**	**NO**
	Devuelve un usuario con el ID pasado por parámetro	
GET	**/users/:id/playlists**	**NO**
	Devuelve una lista con las listas de reproducción de un usuario cuyo ID es pasado por parámetro.	
GET	**/users/:id/tracks**	**NO**
	Devuelve una lista con las canciones de un usuario cuyo ID es pasado por parámetro.	
GET	**/search/playlists/:string**	**NO**
	Devuelve una lista de listas de reproducción cuyo nombre contiene la cadena pasada por parámetro.	
GET	**/search/users/:string**	**NO**
	Devuelve una lista de usuarios cuyo nombre contiene la cadena pasada por parámetro. Si la cadena empieza por @ la búsqueda se realiza por nombre de usuario y no por nombre.	
GET	**/search/tracks/:string**	**NO**

	Devuelve una lista de canciones cuyo nombre contiene la cadena pasada por parámetro.	
GET	**/me**	**SI**
	Devuelve los datos del usuario logueado, menos la contraseña.	
PUT	**/me**	**SI**
	Actualiza los datos del usuario logueado. Los datos que se deseen cambiar se pasan por el cuerpo de la petición en formato JSON.	
POST	**/me/upload**	**SI**
	Utilizada para subir las imagenes de perfil y de fondo del perfil.	
GET	**/me/playlists**	**SI**
	Devuelve las listas de reproducción del usuario logueado.	
POST	**/me/playlist**	**SI**
	Crea una nueva lista de reproducción**.** Los datos como el nombre, descripción, etc. se pasan por el cuerpo de la petición en formato JSON.	
PUT	**/me/playlist/:id**	**SI**
	Actualiza los datos de una lista de reproducción del usuario logueado cuyo ID es pasado por parámetro. Los datos como el nombre, descripción, etc. se pasan por el cuerpo de la petición en formato JSON.	
PUT	**/me/playlist/:id/add-track**	**SI**
	Añade una canción a una lista de reproducción cuyo ID es pasado por parámetro. El identificador de la canción se pasa por el cuerpo de la petición en formato JSON.	
POST	**/me/playlist/:id/uploadImage**	**SI**
	Sube y añade la imagen subida a una lista de reproducción cuyo nombre es pasado por el parámetro id.	
DELETE	**/me/playlist/:id**	**SI**
	Elimina una lista de reproducción cuyo ID es pasado por parámetro.	
GET	**/me/tracks**	**SI**
	Devuelve una lista con las canciones creadas por el usuario logueado.	
POST	**/me/track**	**SI**
	Crea una nueva canción. El documento creado solo tendrá un identificador. El identificador será devuelto si es creada con éxito.	
PUT	**/me/track/:id**	**SI**
	Actualiza los datos de una canción cuyo ID es pasado por parámetro. Los datos para actualizar son pasados por el cuerpo de la petición.	
POST	**/me/track/:id/uploadAudio**	**SI**
	Sube el audio al servidor para una canción cuyo ID es pasado por parámetro.	
POST	**/me/track/:id/uploadImage**	**SI**

	Sube una imagen al servidor para una canción cuyo ID es pasado por parámetro.	
DELETE	**/me/track/:id**	**SI**
	Borra una canción cuyo ID es pasado por parámetro.	
GET	**/stream/:id/dash/:file**	**NO**
	Este endpoint lo utilizaremos para obtener los segmentos de audio para el streaming basado en MPEG-DASH. Más detalles en el punto 3.3.6.	
GET	**/stream/:id/html5**	**NO**
	Este endpoint se puede utilizar para realizar streaming que funciona con HTML5. Se puede añadir este endpoint a un tag <audio> de HTML para comenzar a escuchar el audio.	

Tabla 3-1: Endpoints de la API REST.

3.3.5 Subida y procesamiento de archivos

Los archivos subidos se guardan dentro del directorio /storage, dentro de la raíz de la aplicación. Las imágenes irán al directorio /storage/images y el audio a /storage/audio. Dentro de cada directorio de audio e imágenes, se creará otro directorio por cada canción o lista de reproducción, con nombre el id de la canción o la lista de reproducción, para que sea rápida el acceso a los archivos. Dentro de cada directorio se guardará el archivo original, con el nombre 'original' y la extensión correspondiente (.jpeg, .png, .mp3).

Las imágenes serán procesadas utilizando una librería para Node llamada Sharp, que al estar escrita en C++ nos brinda un gran rendimiento. Las imágenes serán recortadas en forma de cuadrado y centradas. Los tamaños de salida serán de 64x64, 128x128, 256x256, 512x512, 1024x1024 y 2048x2048 píxeles, el tamaño máximo dependerá del tamaño de la imagen original. Al tener variedad de dimensiones podemos dar a elegir a los desarrolladores de los clientes que tamaño de imagen prefieren para cada caso. Simplemente utilizando imágenes con el mejor tamaño que se adapte a las dimensiones del contenedor en el que se muestran se incrementa considerablemente el rendimiento de las aplicaciones, dependiendo del dispositivo en el que se utilice y de si la aplicación es nativa o web se notará más o menos.

En mis pruebas con la aplicación cliente que veremos más adelante, en un principio utilizaba las imágenes originales para cualquier tamaño de contenedor. En un menú lateral en el que se muestra la cola de reproducción, la imagen de cada canción se muestra en un contenedor de 40x40 píxeles, y este menú se puede ocultar y mostrar pulsando un botón, deslizándose dentro y fuera de la pantalla con una animación de 0.3 segundos. Al utilizar las imágenes originales, el mostrar y ocultar el menú en un móvil duraba alrededor de 3 segundos y la animación iba a trompicones. Tras cambiar a imágenes de 64x64 y 128x128 pixeles, el menú se muestra en menos de 1 segundo y de forma más o menos fluida.

También otro procesamiento que hago, no tan importante como el anterior, pues es solo para fines estéticos, es utilizar otras dos librerías para extraer el color dominante de la imagen y también detectar qué color de texto se ve mejor con el color dominante extraído. Esta extracción de colores también se puede hacer en el cliente, pero tras comprobar que, con

varias imágenes en pantalla, mientras se procesaban, la aplicación se ralentizaba, decidí mover este proceso al servidor.

Para el procesamiento del audio he utilizado el conocido framework FFmpeg y la herramienta MP4Box que está dentro del framework de GPAC. El proceso es parecido al de las imágenes. Se sube el audio y se guarda en el directorio /storage/audio/<id-de-la-canción>/ con el nombre original y extensión .mp3. En una primera versión llamaba a FFmpeg y MP4Box utilizando solo Node. El código quedaba un poco feo y largo debido a que las funciones son asíncronas e iba metiendo las siguientes funciones que necesitaba dentro del callback de la anterior. La mejor solución que se me ocurrió fue crear un script con bash que se encarga de hacer las llamadas a FFmpeg y MP4Box, de esta forma en Node solo tengo que hacer una llamada a este script.

El funcionamiento del script es sencillo. Primero obtengo el bitrate del audio utilizando FFprobe, dependiendo del bitrate del audio original voy añadiendo a una lista los bitrates a los que se puede exportar. Con los bitrates a los que se puede exportar me refiero a bitrates menores o iguales que el del audio original, pues, aunque el bitrate de la canción original sea por ejemplo de 128kbps, FFmpeg te permite convertirlo a otro bitrate mayor, pero no tiene sentido hacer esto pues no se puede crear más calidad de donde no la hay. Los posibles bitrates de salida son 128kbps, 192kbps, 256kbps y 320kbps. Una vez que se eligen los bitrates utilizo FFmpeg para crear varias copias del archivo original cada una con un bitrate de los elegidos. Además, el formato de salida de estos archivos cambia de mp3 a m4a. El cambio de formato se debe a que la librería del cliente oficial de MPEG-DASH solo soporta contenedores mp4 y m4a es el contenedor mp4 de audio. Una vez que se hayan creado las distintas versiones del audio original, utilizando MP4Box se segmentan estos archivos y se guardan en un directorio /dash en la misma localización que el audio original.

Esta herramienta también se encarga de crear el archivo index.mpd que es el archivo que primero se envía al cliente para que conozca qué calidades existen para elegir y la localización de estos segmentos. Para este trabajo he decido crear segmentos de 4 segundos.

Otro detalle a comentar es que he añadido un filtro en la subida que solo permitirá subir imágenes en formato .jpg o .png y audio en .mp3.Tampoco se permite subir imágenes de más de 2 MB o audios de más de 300 MB. Puede sorprender el límite de 300 MB para el audio, la razón es que no solo se permiten subir canciones, si se quiere se pueden subir mixes o podcast, que suelen tener una duración de entre 20 minutos y 1 o 2 horas.

3.3.6 Streaming de música

Como ya hemos visto en el punto 3.3.4, utilizaremos dos endpoints para realizar streaming:

```
/stream/:id/html5
/stream/:id/dash/:file
```

Con el primer endpoint el streaming se hace utilizando una librería que implementa el estándar HTML5. Esta librería funciona básicamente como un servidor de archivos estáticos, pero tiene implementada la funcionalidad de solicitud por rangos de bytes. Esto nos permite en el cliente cambiar el punto temporal de la reproducción sin tener que esperar a que se

descargue el audio hasta el punto que queremos reproducir. HTML5 también irá descargando el archivo desde el punto temporal que indiquemos hasta el final, y se permite cambiar de punto temporal cuando queramos. En este tipo de streaming el archivo del que se hace stream es el archivo original de audio.

La forma de hacer streaming anterior funciona bien y nos puede permitir tener un primer punto de comienzo para ir realizando pruebas en el desarrollo. Para mejorar la calidez y rapidez del streaming necesitamos implementar otro método mejor.

Como ya se ha nombrado alguna vez más, utilizaremos el estándar MPEG-DASH para conseguir nuestro objetivo. Los reproductores que soportan MPEG-DASH, como por ejemplo el oficial, dash.js, se encargan de solicitar los segmentos según sea necesario y además van almacenando varios segmentos en un buffer dinámico, sin descargarse el audio entero como pasa utilizando solo HTML5, pues la mayoría de veces las canciones no se escuchan enteras, o se escuchan unos pocos segundos y se pasa a la siguiente canción. Además, son capaces de adaptarse a la velocidad de red eligiendo la calidad máxima que permita reproducir sin cortes o con el menor número de cortes el audio. Con buffer dinámico me refiero a un buffer que va variando con el tiempo de tamaño y que además es capaz de ir cambiando segmentos de menor a mayor calidad o viceversa, según considere mejor. Los segmentos también se ofrecerán como archivos estáticos, pero sin poder elegir rangos.

Aunque descargar el audio original o los segmentos es rápido utilizando Node, para un trabajo futuro se tendría que considerar utilizar un servidor dedicado a ofrecer archivos estáticos, como por ejemplo NGINX.

3.3.7 Otros detalles

3.3.7.1 Búsqueda simple

Para dotar a nuestra aplicación de un pequeño motor de búsqueda haremos uso de la capacidad de MongoDB de realizar búsquedas mediante expresiones regulares. Para ello tenemos tres endpoints, indicados en el punto 3.3.4, que nos permitirán buscar canciones, listas de reproducción o usuarios por el nombre. Para los usuarios, si utilizamos el carácter @ como primer carácter, la búsqueda se realizará por nombre de usuario y no por nombre. La búsqueda consistirá en coger los documentos que contengan en el nombre los caracteres de la búsqueda. Por ejemplo, si tenemos tres usuarios con nombres, Aabc, Bddde, Abee, si buscamos con la cadena ‘ab’, nos devolverá el primer y el tercer usuario.

3.4 Cliente

3.4.1 Visión general

El cliente desarrollado para este trabajo es una aplicación web de una solo página que puede ser visualizado en un navegador web. Al ser de una sola página al pulsar en un botón de navegación, en vez de pedir de nuevo toda la página al servidor con la nueva página solicitada, solo se modifica una parte de la página. En este caso, como la aplicación se ha desarrollado usando Angular, cuando cambiamos de página solo pedimos los datos que queremos mostrar al servidor explicado en el punto anterior y la aplicación de Angular se encarga de visualizar esos datos con los componentes que ya tiene descargados, sin recargar la página.

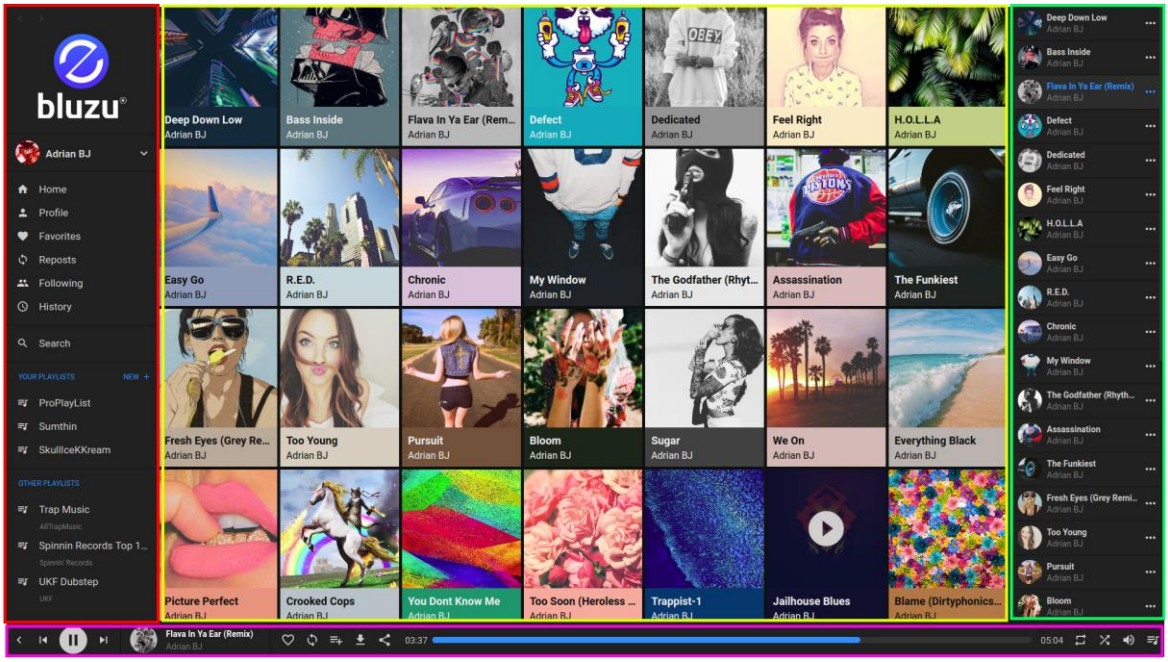


Figura 3-12: Partes visuales principales de la aplicación.

La aplicación tiene 4 partes principales que como puede verse en la figura 3-11, están separadas por colores. A la izquierda, en color rojo, tenemos un panel que muestra un menú de navegación, más alguna opción más. A la derecha, de color verde, tenemos un panel que muestra las canciones que se encuentran en la cola de reproducción y cuál está siendo reproducida de todas ellas. Abajo, en color rosa, tenemos un panel que muestra un reproductor, con la canción que se está reproduciendo, botones para play y pause, cambiar el punto temporal de la canción en reproducción, etc. Por último, en el centro, de color amarillo, tenemos un panel que mostrará el contenido, y es el que va cambiando cuando se cambia de URL, el resto siempre se muestran y la página no se recarga. En esta sección central se mostrarán páginas como, la principal o de bienvenida, la de perfil de los usuarios, búsqueda, listas de reproducción, subida de canciones, etc.

Recordando un poco la explicación de Angular del punto 3.2.6, en Angular las vistas se llaman componentes, y están compuestas por una clase de JavaScript, una plantilla HTML y otra CSS. La clase se encarga de controlar el componente y darle funcionalidad.

Si volvemos a mirar a la figura 3-11, cada parte separada por un color es un componente distinto. Estos componentes son los más grandes que tenemos en la aplicación, pues estos componentes, y otros, tienen otros componentes más simples dentro. Por ejemplo, fijándonos en la figura 3-12, podemos ver la página de perfil de un usuario, el componente que muestra la página del usuario está en amarillo y dentro de este componente tenemos otros componentes que se mostrarán en la zona azul claro. Los componentes de la zona azul claro muestran la página principal del usuario, canciones, álbumes y listas de reproducción que ha creado, información sobre él, etc. Estos componentes se irán mostrando u ocultando dependiendo de qué botón de la barra de navegación de color negro se pulse. El resto de la página no cambiará, solo la zona azul.

Los componentes pueden ser tan grandes, como estos, o más simples, como por ejemplo, en la cola de reproducción, que se muestra en el componente de la derecha, cada canción mostrada podría ser un componente, o incluso podemos crear componentes más simples, como un botón. Tenemos libertad para elegir lo que más se adapte a lo que queremos conseguir, por eso podemos hacer que cada elemento visual sea un componente o simplemente utilizar HTML.

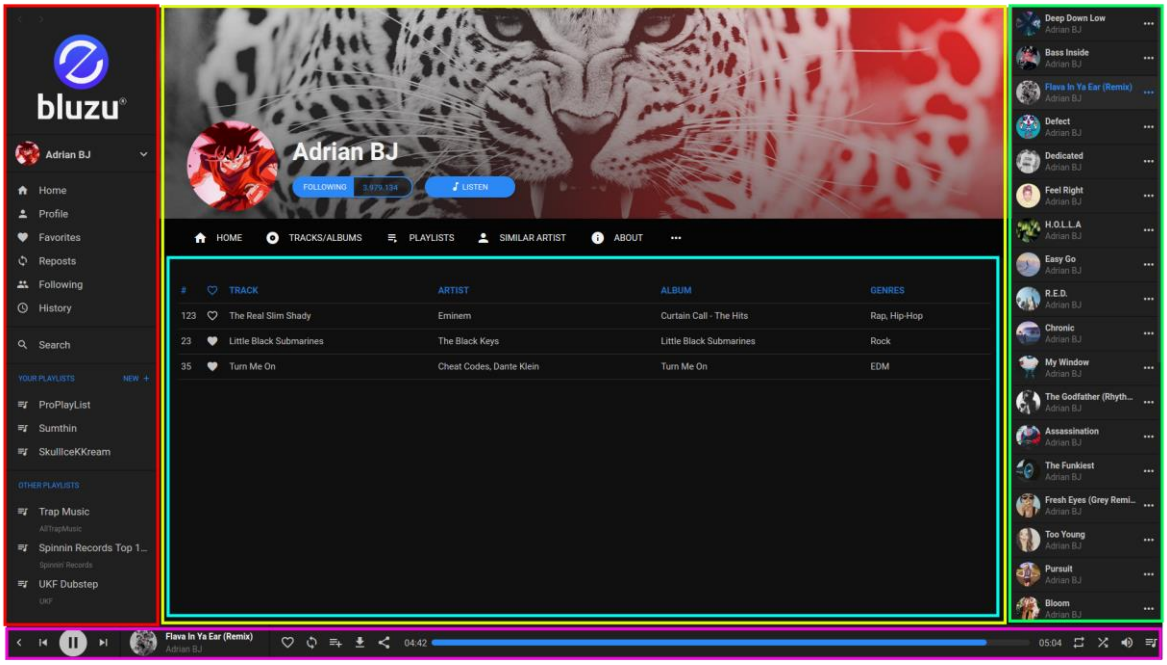


Figura 3-13: Partes visuales principales de la aplicación más de perfil de usuario.

Otro ejemplo de componentes podemos verlo en la figura 3-13. En esta figura se muestra el componente principal que usaremos para mostrar formularios para subir canciones. En este caso, cada formulario (formado por todo el rectángulo gris) sí que ha sido creado como un componente, porque al hacerlo así es mucho más fácil obtener los datos de cada formulario, pues esta página puede ir añadiendo más formularios de forma dinámica pulsando el botón de abajo a la izquierda con texto ‘Add 1 more track’.

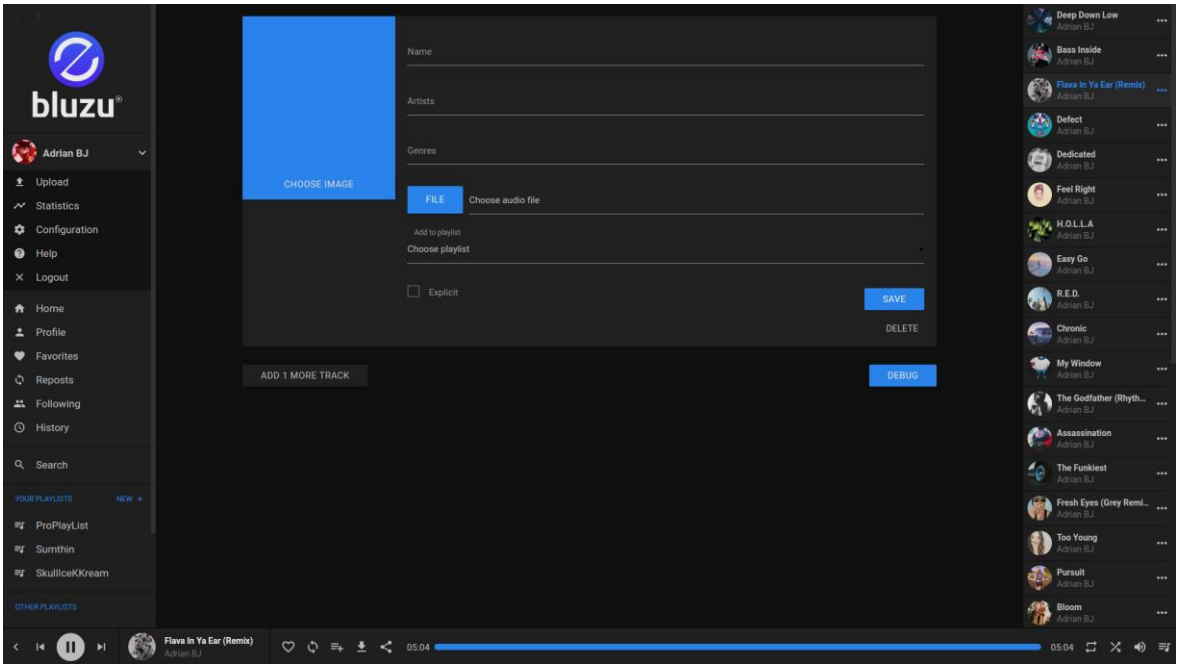


Figura 3-14: Componente principal para subida de archivos.

Otro objetivo de la aplicación era intentar hacer un todo en uno, haciendo que se adaptara lo mejor posible a cualquier pantalla de cualquier dispositivo. Para ello ha sido necesario bastante tiempo para realizar los CSS y algo de JavaScript y HTML. Los componentes laterales se pueden ocultar, solo uno o los dos. El componente de abajo, que contiene el reproductor, se puede ampliar hasta ocupar toda la pantalla. Cuando este componente se amplía al máximo es cambiado por otro con otro diseño. Esto viene bien para los móviles, porque no se pueden mostrar todas las funciones del reproductor en una barra tan pequeña. En la primera imagen de la figura 3-15 se puede ver cómo quedaría en un móvil.

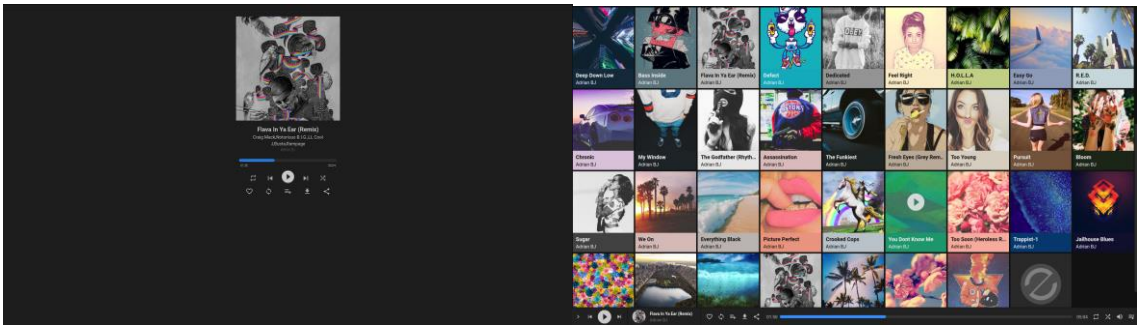


Figura 3-15: Otros ejemplos de vistas de la aplicación. La imagen de la izquierda muestra el componente con el reproductor a pantalla completa en un navegador de escritorio. La imagen de la derecha muestra la aplicación con los componentes laterales ocultos.

Además de adaptarlo a los tamaños de pantalla, también se le ha añadido soporte táctil a algún componente. Los componentes laterales se pueden ocultar haciendo un deslizamiento horizontal sobre ellos. El componente con el reproductor se puede hacer un deslizamiento vertical para ampliarlo u ocultarlo, al igual que hacen otras aplicaciones de música nativas en los móviles.

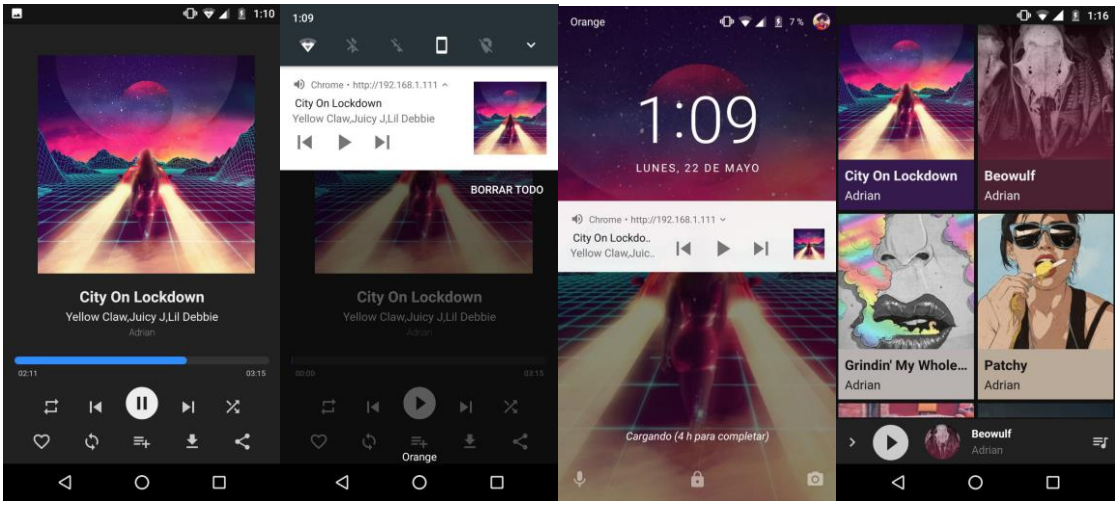


Figura 3-16: Ejemplo de cómo los componentes se adaptan al móvil.

Otro detalle para darle más apariencia de aplicación en los móviles (aun siendo ejecutada en un navegador) es hacer uso de la API Media Session. Pocos navegadores tienen soporte para esta API por el momento. Chrome, desde la versión 57 lo soporta. Esta API nos permite personalizar una notificación que muestra la canción que se está reproduciendo más unos controles básicos. También en algún navegador, como Chrome en Android, existe una opción para añadir las páginas web a la pantalla de inicio del móvil (donde se encuentran los iconos de las aplicaciones). Podemos hacer uso de un archivo manifest.json, que utiliza Chrome al añadir la aplicación a la pantalla de inicio, en el que podemos definir el nombre de la aplicación, iconos de varios tamaños, o incluso pedir que se oculte la barra de navegación con la URL, para darle todavía más un aspecto de aplicación, como se puede observar en la figura 3-15.

3.4.2 Módulos y componentes

Nuestra aplicación está contenida dentro de un módulo. Los módulos pueden contener componentes, directivas, servicios y otras clases y funciones de JavaScript. Todos los servicios, directivas y componentes que proporcionan las librerías de Angular están separadas también por módulos, cada módulo pensado para albergar un tipo de funcionalidad. Nosotros también podríamos separar nuestros componentes por módulos, pero como tampoco es tan grande la aplicación he decidido que todo esté en un solo modulo. A continuación, listare los componentes que he creado:

app
Componente raíz del que cuelgan el resto de componentes. En el defino las 4 partes (explicadas en el punto 3.4.1) donde irán los componentes.
home
Muestra la pantalla de inicio de la aplicación. En esta pantalla podemos ver las canciones y listas de reproducción creadas por todos los usuarios.
play-queue
Muestra la lista de canciones que se encuentran en la cola de reproducción.

player

Parte visual del reproductor que permite reproducir, pausar,pasar a la siguiente canción, cambiar el punto temporal de la canción, etc.

player-big

Misma funcionalidad que el componente player, pero para pantalla completa. Está pensado para su utilización en móviles.

playlist

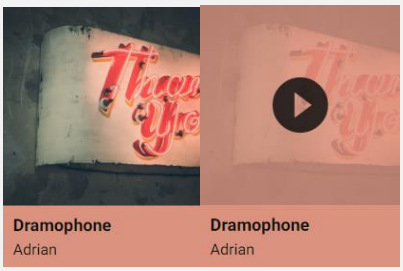
Componente que muestra toda la información de una lista de reproducción junto con una lista de las canciones que tiene.

search

Muestra una barra de búsqueda y debajo los resultados de la búsqueda. Se pueden buscar usuarios, canciones y listas de reproducción.

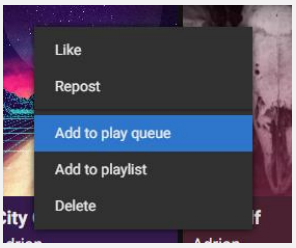
box

Muestra en una caja tanto los usuarios, como las canciones o las listas de reproducción. Cuando el ratón se pone encima del componente se muestra una opción de reproducir la canción o canciones ese elemento.



context-menu

Muestra una lista de opciones al hacer click con el botón derecho del ratón en algún componente.



edit-playlist-form

Muestra un modal que permite crear o modificar una lista de reproducción.

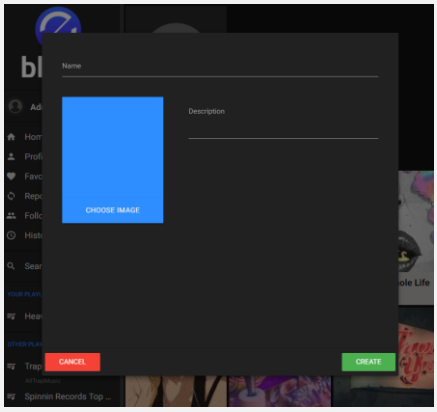

edit-track-form
Muestra un modal que permite crear o modificar una canción.
sidenav
Menú lateral izquierdo utilizado para la navegación.
upload
Muestra uno o varios formularios para subir canciones. Puede verse en la figura 3-13, en el panel central.
upload-track
Muestra un formulario para subir una canción. Puede verse en la figura 3-13, en el panel central. Cada formulario es un componente de este tipo.
user
Muestra la información del usuario. Contiene un menú de navegación para cambiar entre los distintos componentes que muestran información del usuario. Puede verse en la figura 3-12.
user-about
Componente que se muestra dentro del componente user y que muestra información que el usuario haya dado sobre él.
user-home
Componente que muestra la actividad del usuario, como canciones recientes que ha subido, canciones que ha añadido a alguna lista, etc...
user-music
Muestra las canciones y los álbumes del usuario.
user-playlists
Muestra las listas de reproducción del usuario.

Tabla 3-2: Componentes creados.

3.4.3 Servicios

Los servicios son clases que tienen un propósito bien definido, deben hacer algo específico y hacerlo bien. Además, tenemos que tener en cuenta que los componentes se organizan en

forma de árbol, y los nodos hijo no pueden enviar información al padre directamente, pues la información solo puede fluir de los nodos padre a los hijos. Los servicios pueden ayudarnos también a comunicar distintos componentes. Al igual que los componentes, los servicios solo pueden accederse desde el componente que declara el servicio y los componentes hijos. Para tener un acceso desde todos los componentes, los servicios pueden declararse en la definición del módulo o en el componente raíz. Los servicios suelen tener solo una instancia. Tanto los servicios como los componentes pueden lanzar eventos que pueden ser capturados por otros servicios y componentes. Los eventos pueden ser lanzados junto a un objeto. Yo solo lanzo eventos en los servicios, pues es donde me han resultado más útil lanzarlos.

Para este trabajo tenemos 3 servicios:

bluzu
He llamado a mi aplicación (cliente y servidor) con el nombre de 'Bluzu', y por eso este servicio tiene ese nombre. Su funcionalidad consiste en hacer las peticiones al servidor y devolver esa información al componente que la solicite. Utiliza la API definida en el punto 3.3.4. También guarda la información de autenticación del usuario. Lanza varios eventos para indicar cuando el usuario ha hecho login o logout y si se ha creado o eliminado una canción o lista de reproducción.
global
Utilizo este servicio para poder enviar información a cualquier componente. Solo tiene dos funciones en este momento, una es enviar información al componente context-menu para indicarle que debe mostrar en la lista de opciones y la segunda función es para enviar el tamaño horizontal del panel central al resto de componentes que lo necesitan para adaptar su tamaño, en nuestro caso solo el componente box utiliza esta función. Lanza dos eventos, uno para indicarle al context-menu que tiene que mostrarse y otro para indicar que el tamaño del panel central ha cambiado. La razón por la que utilizo este servicio para enviar el tamaño del panel central a otros componentes se debe a que es la mejor forma de hacerlo, pues se puede obtener el tamaño de la ventana del navegador y capturar el evento que se lanza cuando cambia de tamaño, pero el navegador no lanza eventos cuando cambia el tamaño de un elemento HTML. El tamaño del panel central cambia cuando se cambia el tamaño de la ventana del navegador y cuando se muestra u oculta alguno de los paneles laterales. Se puede conseguir que los componentes se adapten solo utilizando CSS, pero el resultado no era siempre bueno.
player
Este servicio se encarga de reproducir las canciones, tanto utilizando HTML5 como utilizando la librería dash.js para el streaming adaptativo. También contiene la lista de canciones de la cola de reproducción. Todos los componentes deben (o deberían) de utilizar este servicio para controlar la reproducción y no crear otros objetos de audio de HTML5. Lanza tres eventos: uno cuando se cambia de canción, otro cuando hay algún cambio en la cola de reproducción y otro para indicar que ha terminado de reproducirse la canción en reproducción. También se encarga de actualizar la notificación en los móviles que puede verse en la figura 3-15.

Tabla 3-3: Servicios creados.

3.4.4 Streaming

El streaming en el cliente es fácil de realizar debido a que contamos con la posibilidad de utilizar alguna librería que nos haga el trabajo. Para el streaming en HTML5 no hace falta ninguna librería externa. Para el streaming MPEG-DASH utilizaremos la librería oficial

dash.js. Esta librería se asocia a un componente de audio o video de HTML5 y se encarga de coger los segmentos y hacer que parezcan uno solo. Al asociarse con un componente de HTML5 podemos controlar la reproducción haciendo uso de la API de HTML5 para media para ambos tipos de streaming.

4 Desarrollo

En este punto ya podemos tener una idea de cómo funciona tanto el servidor como el cliente. Ahora vamos a ver cómo construirlos. Debido al límite de hojas establecido para esta memoria, solo explicaré lo fundamental. No mostraré, por ejemplo, nada de CSS, en el caso del cliente.

Antes de empezar a desarrollar tendremos que tener instalado Node.js, tanto para el cliente como el servidor. Para el servidor también necesitamos instalar MongoDB y Redis.

Como vamos a utilizar Node, lo primero que necesitamos crear es un fichero package.json en la raíz de cada uno de nuestros dos proyectos (cliente y servidor). Este fichero contiene metadatos y las dependencias de nuestra aplicación. Quiero destacar las propiedades 'scripts', 'dependencies' y 'devDependencies'. En 'scripts' podemos poner comandos con argumentos. Para ejecutar un comando tan solo tenemos que escribir en la terminal 'npm run <nombre-comando>'. Para el comando 'start' sólo es necesario escribir 'npm start'. El comando 'start' es el estándar para iniciar nuestra aplicación. En estos comandos podemos hacer uso de paquetes instalados en el proyecto. En 'dependencies' se encontrarán las dependencias de nuestra aplicación. Podemos instalar más paquetes escribiendo en la terminal el comando 'npm install --save <nombre-dependencia>'. En 'devDependencies' se encuentran dependencias necesarias solo para el desarrollo. Como vamos a usar TypeScript en vez de JavaScript, necesitamos tener instalado el compilador de TypeScript. Podemos instalarlo globalmente en nuestro ordenador escribiendo en la terminal 'npm install -g typescript' o podemos añadirlo a nuestras dependencias de desarrollo escribiendo 'npm install --save-dev typescript'. Para algunos paquetes nos puede venir bien instalar los tipos que definen las funciones/clases/objetos del paquete, estos tipos los utilizará el compilador de TypeScript para comprobar que estamos utilizándolos bien. Los paquetes de tipos empiezan por '@types/'.

Otro archivo de configuración necesario es tsconfig.json. Este archivo es leído por el compilador de TypeScript cada vez que se compila el código. TypeScript permite compilar a distintas versiones de JavaScript, en este fichero podemos definir a cuál queremos que compile. Este archivo se encontrará también en la raíz de nuestros proyectos.

Gran parte del código explicado podrá encontrarse en el anexo E. Para hacerse una idea de los ficheros package.json y tsconfig.json de cada proyecto, y las dependencias de cada uno, podéis ir a este anexo.

En el anexo C también se encuentra la estructura de directorios y los archivos que tienen las aplicaciones desarrolladas.

Antes de empezar a escribir código recomiendo copiar los package.json e instalar las dependencias indicadas en ellos con el comando 'npm install'.

NOTA: Cuando nombre un archivo, si contiene dos asteriscos al final del nombre (<nombre>**), significa que el código entero o una parte se encuentra en el anexo E.

4.1 Servidor

Comenzaremos creando un directorio src/ en la raíz del proyecto. Será donde pondremos todo el código fuente. Dentro de este directorio crearemos 7 directorios más: auth, models, ssl, other, public, routes y views. En **auth** tendremos código dedicado al proceso de autenticación, en **models** estarán definidos nuestros esquemas y modelos para la base de datos, en **ssl** se encontrarán el certificado y la clave para HTTPS además de un script para generarlos, en **other** pondremos código que no encaje en otro directorio, en **public** pondremos ficheros CSS, JavaScript, fuentes e imágenes que queramos utilizar con las páginas HTML que muestre el servidor, en **routes** se encontrarán las clases que definen los endpoints de la API REST y finalmente en **views** tendremos nuestras páginas en HTML o EJS (plantillas JavaScript).

Ya tenemos la estructura de directorios. Continuaremos creando los ficheros principales.

Crearemos primero **server.ts****, que contendrá el código necesario para crear un servidor HTTP o HTTPS. Necesitamos importar los módulos de Node nativos de http, https, fs, cluster y os. Con el módulo **http** se puede crear un servidor HTTP, con **https** un servidor HTTPS, con **fs** podemos leer el certificado y la clave SSL necesarios para crear el servidor HTTPS, con **cluster** podremos hacer uno o varios forks para tener varias instancias del servidor, con **os** podemos obtener cuantos procesadores tiene nuestro ordenador, pues he decidido crear tantas instancias como procesadores tengamos. Nuestro servidor es sin estados e independientes los unos de los otros, por lo que si creamos varios procesos hijos no tenemos que preocuparnos de hacer que se comuniquen entre ellos, con hacer fork nos basta. Hay una excepción que contare mas adelante. Tambien importamos unas variables del fichero **global.ts**** y **app.ts****. En **app.ts**** crearemos y configuraremos nuestra aplicación. Cuando creemos el servidor, tendremos que pasar nuestra aplicación a las funciones **http** y **https**.

En **global.ts**** he definido unas variables a las que quiero acceder desde cualquier sitio que lo necesite. Estas variables están inicializadas en este fichero y no se modifican desde ningún otros sitio. En este fichero aprovecho también para crear la conexión a la base de datos utilizando una función definida en **database.ts****.

NOTA: Es común utilizar un archivo **.env** para configuración. Se suele definir en él por ejemplo el puerto que queremos que utilice la aplicación o tokens de acceso a otros servicios externos. Si vamos a conectarnos a un servicio externo los tokens deberían de ir si o si en este fichero, además de que si utilizamos un repositorio público para guardar nuestro código fuente, este archivo no debería de subirse, para proteger nuestros tokens de acceso a los

servicios. En este trabajo no utilizamos servicios externos y por eso no he utilizado este fichero, además de que necesitaba utilizar el módulo **path** para definir unas rutas.

En **app.ts**** crearemos y configuraremos nuestra aplicación servidor haciendo uso de la librería **Express.js**. Necesitaremos importar otras dependencias que utilizara Express para parsear las peticiones que reciba. Después de crear una instancia de Express, procedemos a configurarla. Le vamos diciendo que tiene que utilizar las dependencias que hemos importado, también definimos los directorios /public y /storage/images y /storage/audio como directorios estáticos. Le decimos que utilice la librería ejs para parsear las plantillas en ejs. Después le decimos que utilice la base de datos en memoria Redis para almacenar sesiones, que hemos definido que durarán un máximo de 5 minutos. Por último, habilitamos los CORS para permitir realizar peticiones a este servidor desde otros hosts distintos al host que tiene esta aplicación. También definimos las rutas raíz y que clases se encargaran de esas rutas.

Una de los paquetes añadidos a Express, Morgan, nos muestra por la consola las peticiones al servidor, junto a la URL, el código y el tiempo que tarda en responder el servidor. Puede sernos útil para debuguear.

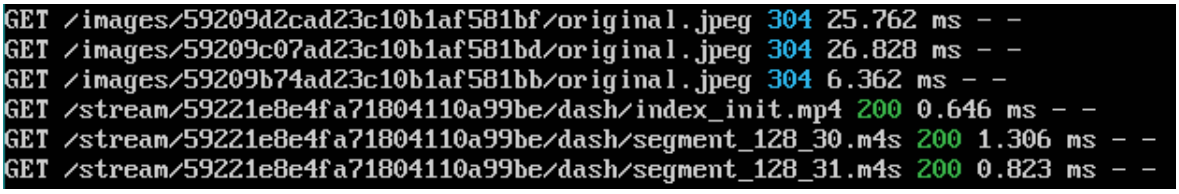


Figura 4-1: Ejemplo de salida de Morgan.

Cuando he explicado el fichero server.ts he dicho que había una excepción a la hora de crear los forks. La excepción ocurre cuando creamos las sesiones que necesitaremos para realizar el proceso de autenticación con OAUTH2, pero gracias a la base de datos Redis, las sesiones se comparten entre todos los hilos, y nosotros no tenemos que preocuparnos de nada.

En **database.ts**** he creado una función que realiza la conexión a la base de datos de MongoDB. Utilizamos la librería de Mongoose para realizar la conexión. Al utilizar la librería de Mongoose para realizar la conexión, Mongoose almacena esta conexión en una variable propia 'connection', por lo que una vez que nos hemos conectado a la base de datos, con que importemos esta librería en otros archivos será necesario para tener acceso a la base de datos. Es como si la conexión fuera una variable global. Esto hace fácil el utilizar solo una base de datos, para conectarse a más de una es más difícil.

En el punto 3.3.3 ya se explicaron los esquemas que se iban a utilizar para guardar los datos en la base de datos. Explicare solo el caso de los usuarios, que pondremos en el fichero **models/user.ts**, pues el resto son iguales. Lo primero que haremos será importar varias cosas: dos tipos y una función de Mongoose, una variable con la URL base de nuestro servidor (https://<ip>:<puerto>/) y por último un paquete de encriptación (bcrypt). Después de las importaciones veremos el esquema explicado anteriormente, a continuación, veremos una función de transformación, que sirve para personalizar cómo se devuelven los usuarios a los clientes, por ejemplo, la utilizamos para borrar la contraseña de los resultados, después definimos una función que se ejecutará antes de guardar un usuario en la base de datos, la

utilizaremos para encriptar la contraseña utilizando bcrypt. Por último, creamos el modelo a partir del esquema definido.

Continuamos ahora con la parte encargada de la autenticación. Para ello necesitamos varias dependencias: passport, passport-local, passport-http, passport-http-bearer y oauth2orize. Con passport nos encargamos de comprobar que los nombres de usuario, IDs de clientes, tokens y las contraseñas son válidas. Para validar a los usuarios utilizaremos la clase Strategy de passport-local, para validar a los clientes utilizaremos la clase BasicStrategy de passport-http, y para validar los tokens de OAUTH2 utilizaremos la clase Strategy de passport-http-bearer. Toda la implementación de estos métodos de autenticación se encontrara en un fichero **auth/strategies.ts**. La librería oauth2orize nos ayudará a implementar el protocolo OAUTH2. El código que utiliza esta librería se encuentra en **auth/oauth2.ts**. Aunque no muestre el código de estos dos ficheros, se podría decir que es fácil escribirlo. El funcionamiento es básicamente acceder a la base de datos para obtener el nombre de usuario y su contraseña, el ID del cliente y su clave secreta y el token de autorización de OAUTH2 y comprobar con las claves, tokens, etc. que vienen en las peticiones. En la documentación de estas librerías se pueden ver ejemplos de cómo se usan.

Por último, veremos donde se configuran los endpoints de la API y su funcionalidad. Para ello vamos a ver el ejemplo más sencillo mirando al fichero **routes/all.router.ts****. Todos los endpoints los he definido dentro del directorio routes. Para recordar que se hace en cada endpoint podéis volver al punto 3.3.4. En el fichero **all.router.ts** definiremos los endpoints que nos devolverán todas las canciones, listas de reproducción y usuarios del servidor, como comenté en el punto 3.3.4 estos endpoints solo son indicados mientras desarrollamos. He decidido mostrar este ejemplo y no otros porque este es el fichero menos extenso y más fácil de entender. Como podréis observar si miráis el código en el Anexo E, solamente hacemos una búsqueda en la base de datos y devolvemos los resultados al cliente que lo haya pedido. MongoDB nos devuelve los resultados en JSON por lo que podemos devolver los datos directamente. En las listas de reproducción y las canciones tenemos una referencia al usuario que las ha creado, podemos devolver la información de este usuario en vez de solo su ID haciendo uso del método populate. En populate tenemos que indicar que atributo, que apunta a otro documento, queremos cambiar, y luego qué atributos del documento al que apunta queremos coger, en nuestro caso solo cogemos el nombre y el nombre de usuario.

El fichero más extenso es el que se encarga de los endpoints que empiezan por **/me**. Es el más extenso porque se encarga de todo sobre el usuario logueado. Se utiliza también para obtener las canciones y las imágenes subidas por los usuarios y procesarlas. Tiene la misma estructura que **all.router.ts**, pero quería destacar una diferencia en la definición del endpoint y de las funciones que debe utilizar, por ejemplo:

```
this.router.post('/playlist', isTokenValid, this.newPlaylist);
```

La diferencia con respecto al ejemplo de **all.router.ts** es que lo primero que hacemos es llamar a una función **isTokenValid**. Esta función la definimos en **auth/strategies.ts** y sirve para comprobar que el token de OAUTH2 es válido, si no es válido no se continúa a la siguiente función.

Para subir las canciones se ha hecho uso de la librería **multer**, que nos permite definir en qué directorio queremos guardar los archivos subidos y definir filtros, para por ejemplo solo

permitir subir archivos de un formato o limitar su tamaño máximo. Para procesar las imágenes se ha utilizado la librería **sharp**, que nos permite recortar y redimensionar las imágenes, entre otras cosas. Para obtener el color primario de una imagen hemos utilizado **jimp-color-thief**. Todas estas funciones se pueden guardar en un fichero de utilidades distinto al que contiene los endpoints de /me. Para procesar las canciones una vez que se han subido se hace uso de un script que utiliza FFmpeg y MP4Box. Este script lo he llamado **generate-dash.sh**** y he explicado su funcionamiento anteriormente en el punto 3.3.5

4.2 Cliente

Al igual que con el servidor, crearemos un directorio src/ para guardar el código fuente. Dentro de este directorio he definido otros 4: app, css, images, js. En **css**, he puesto unos CSS para sobrescribir unos estilos del framework Materialize (he utilizado este framework para facilitar el diseño visual de la aplicación), en **images** he puesto imágenes como el logo de la aplicación, y en **js** un par de scripts que no pude instalar a través de npm. En **app** se encuentra nuestra aplicación en Angular. Dentro del directorio **app** he creado otros 3 directorios: components, models, y services. En components irán los componentes de nuestra aplicación, en models definiremos las clases de las canciones, usuarios, etc. serán iguales o casi iguales que los definidos en el servidor, en services guardaremos nuestros servicios de acceso global. Los servicios que están asociados a un componente nada mas deberían ir junto a ese componente.

Para explicar el desarrollo de esta aplicación también iré solo a los ficheros fundamentales, sin entrar en mucho detalle. Si no se tienen conocimientos de Angular seguir el desarrollo de la aplicación puede ser difícil. Recuerdo que, aunque Angular es la continuación de AngularJS, los dos son completamente distintos, aunque comparten alguna cosa. La documentación oficial de Angular es muy completa y más o menos fácil de seguir si eres principiante.

Comenzaremos definiendo en **src/** el fichero **index.html****, que tiene el mismo cometido que todas las páginas web. Algunos de los scripts que se añaden a este HTML son: shim.min.js, que contiene polyfills para otros navegadores (los polyfills sirven para llevar funcionalidad de las últimas versiones de JavaScript (ES6) a las más antiguas, como ES5), zone.js, reflect.js y system.src.js, son necesarias para cargar la aplicación de Angular posteriormente, y systemjs.config.js que se encarga de cargar los componentes de nuestra aplicación cuando sean necesarios. También en un script nuestro (dentro de los tags <script>...</script>) tenemos que cargar un script **main.js**** que tenemos que crear nosotros también. Por el final del HTML cargaremos otras librerías como la de DASH, jQuery, Hammer (para eventos táctiles), y Materialize (para diseño estético de la aplicación).

En **main.ts**** tenemos que cargar nuestro módulo que contiene todos los componentes de la aplicación Angular y después llamar a una función de una librería de Angular para cargar este módulo en la página web.

El módulo de nuestra aplicación se define en **app.module.ts****, aquí definimos otros módulos que vamos a utilizar (dentro de imports), nuestros componentes que pueden ser

accedidos desde cualquier otro componente (dentro de declarations), los servicios globales (dentro de providers) y el componente raíz que se mostrará cuando se cargue la aplicación (en bootstrap).

El componente **app.component.ts** es el componente raíz del que cuelgan el resto de componentes de la aplicación. En este componente solo lo he utilizado para dar la estructura visual a la aplicación, definiendo mediante CSS los paneles que explique anteriormente. Dentro de cada panel añado el componente correspondiente. También he puesto aquí la funcionalidad de mostrar y ocultar los paneles laterales y de expandir el panel donde se encuentra el reproductor.

En **app.routing.ts**** he definido las rutas de la aplicación, estas rutas son las que aparecen en la barra del navegador donde aparece la URL. Este fichero es bastante fácil de comprender, podéis ver definidos en un array las rutas de la aplicación junto con el componente que se debe mostrar cuando se esté en esa ruta.

Para mostrar al menos un componente de los desarrollados, he decidido mostrar el componente de búsqueda, **search.component.ts****. Este componente es de los más sencillos de la aplicación. Tiene 4 variables, una lista de listas de reproducción, otra lista de usuarios, otra lista de canciones y la entrada de la barra de búsqueda. Por el constructor le pasamos el servicio que se encarga de hacer las peticiones al servidor. Se ha definido una función que se encarga de utilizar el servicio para pedir los datos y actualizar las listas. En la plantilla de este componente, **search.component.html****, hemos utilizado varias directivas de Angular como son *ngIf, *ngFor. Gracias a estas directivas podemos mostrar fácilmente los resultados de la búsqueda. También hacemos uso de [(ngModel)] en la barra de búsqueda para actualizar la variable searchInput cada vez que cambie el texto introducido en la barra de búsqueda, además también usamos (ngModelChange) junto a una función que será llamada cada vez que se introduzca o se borre una letra de la barra de búsqueda. Las directivas *ngIf y *ngFor responden automáticamente a los cambios de las variables con las que están.

En el anexo he puesto también el servicio **bluzu.service.ts****, no está completo, pero puede dar una idea de cómo funciona. Además de las funciones de búsqueda que utiliza el componente de búsqueda explicado, he dejado otra función de ejemplo, en el que se hace una solicitud PUT y es necesario añadir a la cabecera de la petición el token de autorización. Esta función en concreto muestra una notificación cuando recibe una respuesta, tanto correcta, como de error.

Es importante tener en cuenta que todas las librerías de Angular funcionan de forma asíncrona, y que si se quiere el resultado inmediatamente después de recibir una respuesta del servidor, por poner un ejemplo, es necesario coger el resultado en las funciones de callback. Personalmente, tuve algún error mientras aprendía Angular debido a que no recordaba que todo era asíncrono, e intentaba acceder a variables que estaban vacías o desactualizadas porque las cogía fuera de los callback y entonces accedía a ellas antes de que se actualizara su valor.

5 Integración, pruebas y resultados

Las dos aplicaciones, cliente y servidor se han ido construyendo poco a poco y haciendo pruebas con casi cada función/clase/componente/módulo implementado. Las pruebas han sido manuales y no automáticas. La estructura de las aplicaciones hace que sea fácil añadir nueva funcionalidad, por lo que las pruebas de integración han sido fáciles de realizar.

5.1 Servidor

Para probar el servidor, aparte de hacer pruebas locales, he utilizado otro software: la consola de MongoDB, Insomnia REST y los reproductores de pruebas de dash.js y el proporcionado por GPAC.

He utilizado conjuntamente la consola de MongoDB e Insomnia REST para comprobar que todos los endpoints del servidor funcionaban correctamente. La aplicación de Insomnia es bastante útil para probar APIs REST, viene con todo lo que podemos necesitar para probar nuestra API.

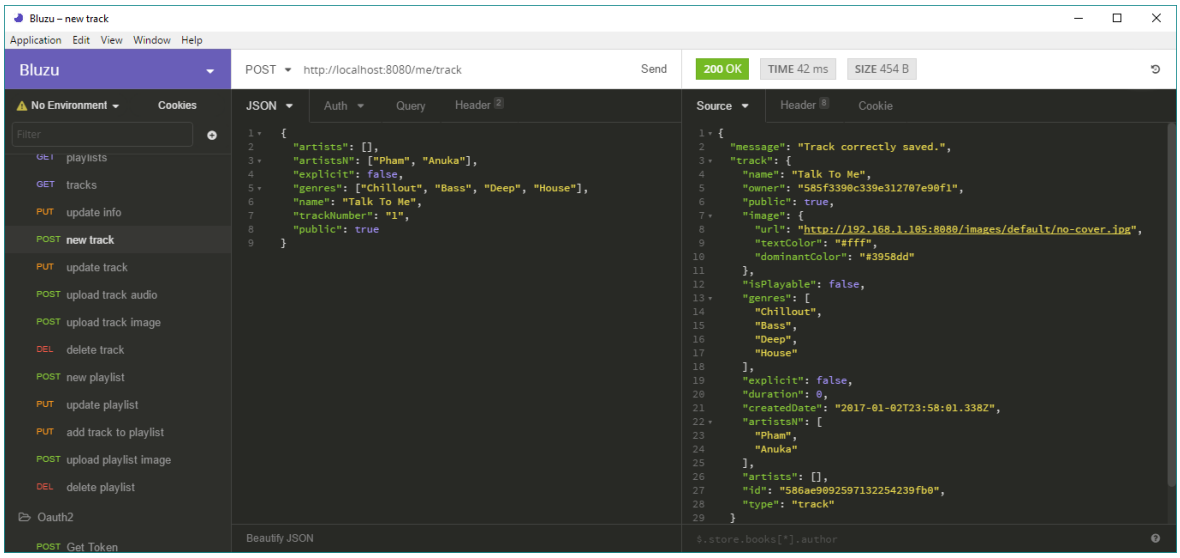


Figura 5-1: Ejemplo de prueba con Insomnia REST.

Podemos ver tres 3 columnas. Empezando por la izquierda, en la primera columna tenemos una lista con las pruebas que vamos creando. Las pruebas se pueden ir agrupando en carpetas. En la segunda columna podemos especificar el tipo de petición, la URL, los parámetros, el cuerpo, autenticación, etc. y en la tercera columna los resultados que devuelve el servidor tras hacerle esa petición.

Para probar que estamos creando bien los segmentos para el streaming adaptativo podemos utilizar el reproductor de pruebas ofrecido por el DASH Industry Forum. Podemos encontrar distintas versiones de este reproductor en: http://dashif.org/reference/players/javascript/ Con este reproductor podemos ver también varias estadísticas como velocidad de descarga, tamaño del buffer, etc.

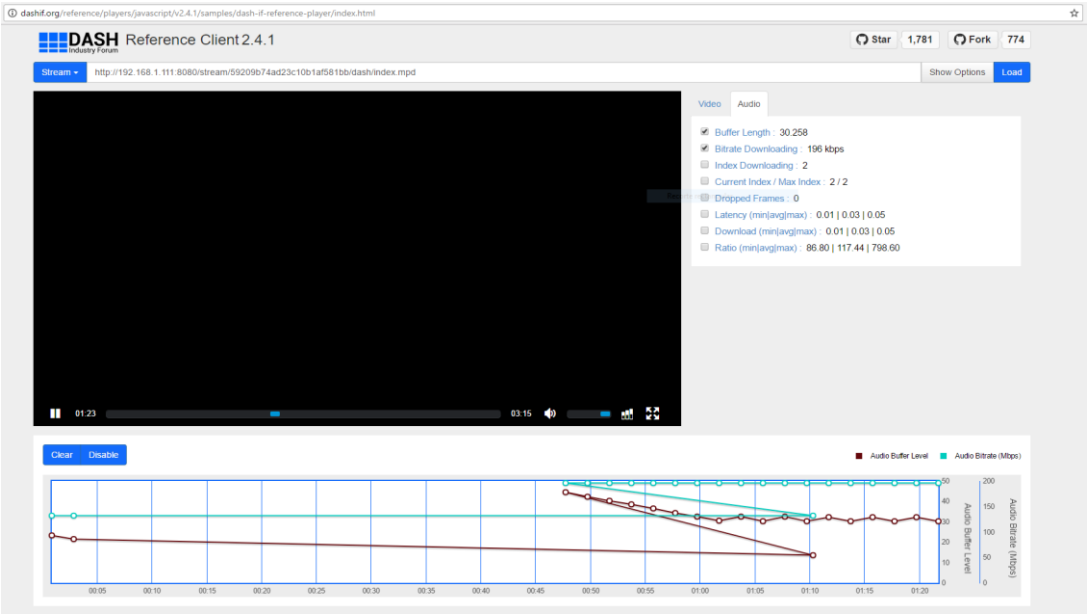


Figura 5-2: Ejemplo de prueba utilizando el reproductor de pruebas de DASHIF.

5.2 Cliente

Para probar las aplicaciones hechas con Angular existen varias herramientas como Mocha o Karma que tiene un funcionamiento parecido a JUnit en Java. Yo preferí hacer pruebas manuales, como ya he comentado. Para cada componente comprobaba que se mostraba en el navegador como quería que se mostrara y que la funcionalidad que tenía que tener era correcta. Una de las cosas que más tiempo me ha llevado ha sido darle estilo a cada componente, y comprobar el estilo solo se puede hacer a mano. Los servicios los probaba normalmente junto a un componente que los utilizaba. Por ejemplo, para probar el servicio de reproducción utilizaba el componente con el reproductor.

6 Conclusiones y trabajo futuro

6.1 Conclusiones

Personalmente pienso que este trabajo ha sido muy útil para mejorar mis conocimientos y destrezas en el desarrollo de aplicaciones Web. Antes de empezar el trabajo, no sabía casi nada de HTML, CSS ni JavaScript, solo podía hacer páginas Web simples con un diseño relativamente feo y escasa funcionalidad. Tampoco había creado ningún servidor, ni ningún servicio. En la universidad había obtenido varios conocimientos relacionados con este trabajo, muchos más bien teóricos, sobre todo en servidores, pero muy poca práctica en general, y sin práctica no se aprende.

Ahora soy capaz de crear páginas/aplicaciones Web mucho más rápido, con más funcionalidad y de más calidad. Además, el crear el servidor por mi cuenta, con toda la funcionalidad que tiene, me ha ayudado a comprender mejor cómo funcionan los servidores

y los servicios. El hacer uno mismo una aplicación más o menos grande es muy útil para fomentar la capacidad de trabajo autónomo, el emprendimiento y la creatividad.

Me ha sorprendido lo rápido que se puede crear cualquier tipo de aplicación con las tecnologías usadas en este trabajo (Node.js, Angular, MongoDB, …) una vez que se tiene experiencia con ellas. Actualmente las uso muy a menudo. También he notado que Node.js puede no ser el más indicado para tareas que requieran de mucho procesamiento, por ejemplo, al subir una imagen y obtener el color principal, el servidor había veces que se ralentizaba un poco y la memoria RAM se incrementaba considerablemente. Esto también puede ser provocado por la librería que he usado, que está escrita completamente en JavaScript (hay librerías de Node.js que usan C o C++ por debajo) debido a que los desarrolladores la escribieron principalmente para su uso en los navegadores.

Me hubiera gustado explicar más detalladamente algún punto, sobre todo en el desarrollo, pero por el límite de hojas no he podido mostrar todo el código realizado y simplemente he explicado lo que he hecho. Si a algún lector le interesa algún tema explicado puede buscar en la web más información, cada vez se pueden encontrar más ejemplos y más información sobre desarrollo con Node.js y Angular.

Si tuviera que volver a realizar este trabajo, volvería a utilizar las mismas tecnologías que he usado ahora, pero realizaría varias mejoras y utilizaría, en el caso del servidor, otros frameworks, lenguajes, etc. junto a Node.js. Para el cliente, en el caso de desarrollo de páginas web, Angular funciona muy rápido y permite crear aplicaciones web complejas en poco tiempo y con relativa facilidad, por el momento no cambiaría este framework por otro.

Por último, destacar que el código desarrollado para poder escribir este trabajo esta disponible en un repositorio público de Git. El enlace al repositorio se encuentra al principio del Anexo D (Código).

6.2 Trabajo futuro

Aunque la aplicación funciona satisfactoriamente, todavía queda amplio margen para la mejora, no solo mejorando la funcionalidad ya existente, sino agregando nueva también.

Para mejorar lo que ya tenemos podemos implementar todos los flujos de autenticación de OAUTH2, mejorando de esta forma la seguridad de nuestro servicio. También, en vez de hacer que un servidor realice todo el trabajo, lo mejor sería repartirlo en varios servidores, cada uno dedicado a una tarea específica. Por ejemplo, podemos tener un servidor dedicado a la autenticación, otro a la API REST, otro para el procesamiento de las canciones e imágenes, otro para almacenamiento, etc. Al dividirlos por tareas específicas nos será más fácil clusterizar cada servicio y también mantenerlos. También se podría mejorar cada servicio utilizando otras tecnologías y lenguajes, quizás para procesar datos, imágenes o canciones nos interesaría crear algún servidor con procesos batch. Adicionalmente, si queremos realizar transacciones en nuestro servicio, necesitamos cambiar a otra base de datos relacional para guardar algún tipo de dato en concreto que necesite de esa característica que no ofrece MongoDB.

Para ampliar funcionalidad se me ocurren varias ideas. Podemos darle cierta similitud a una red social haciendo que los usuarios puedan seguir a otros usuarios y recibir notificaciones cuando alguno suba una canción. También podemos crear clientes nativos o semi-nativos para móviles, para mejorar el rendimiento en ellos. Se puede crear algún sistema de recomendación de canciones y usuarios como hacen todos los servicios de música conocidos. También podemos implementar un sistema de búsqueda mejor que la búsqueda con expresiones regulares que hemos realizado utilizando por ejemplo Elasticsearch. También podemos hacer uso de servicios externos (y de pago) para, por ejemplo, quitarnos el problema del almacenamiento, por ejemplo, utilizando el servicio S3 de Amazon.

Referencias

[1] Página oficial Node.js. https://nodejs.org
[2] Información sobre Node.js en Wikipedia. https://es.wikipedia.org/wiki/Node.js
[3] IBM depeveloperWorks. https://www.ibm.com/developerworks/ssa/opensource/library/os-nodejs/
[4] Página oficial de Redis. https://redis.io
[5] Información sobre Redis en Wikipedia. https://es.wikipedia.org/wiki/Redis
[6] Página oficial de MongoDB. https://www.mongodb.com
[7] Información sobre MongoDB en Wikipedia. https://es.wikipedia.org/wiki/MongoDB
[8] Genbeta Dev. MongoDB. https://www.genbetadev.com/bases-de-datos/mongodb-que-es-como-funciona-y-cuando-podemos-usarlo-o-no
[9] Microsoft. NoSQL vs SQL. https://docs.microsoft.com/es-es/azure/documentdb/documentdb-nosql-vs-sql
[10] DASH Industry Forum. http://dashif.org/about/
[11] Bitmovin. MPEG-DASH vs HLS vs MSS vs HDS. https://bitmovin.com/mpeg-dash-vs-apple-hls-vs-microsoft-smooth-streaming-vs-adobe-hds
[12] Encoding. MPEG-DASH. https://www.encoding.com/mpeg-dash/
[13] Informacion sobre MPEG-DASH en Wikipedia. https://en.wikipedia.org/wiki/Dynamic_Adaptive_Streaming_over_HTTP
[14] Spotify Developers. Web API Authorization Guide. https://developer.spotify.com/web-api/authorization-guide/
[15] DigitalOcean. An Introduction to OAUTH 2. https://www.digitalocean.com/community/tutorials/an-introduction-to-oauth-2
[16] The Game of Code. Conceptos básicos de OAUTH2. http://www.thegameofcode.com/2012/07/conceptos-basicos-de-oauth2.html
[17] Aaron Parecki. Oauth2 Simplified. https://aaronparecki.com/oauth-2-simplified/
[18] Página oficial de Angular. https://angular.io/
[19] Rangle's Angular Training Book. https://angular-2-training-book.rangle.io/
[20] Spotify. https://www.spotify.com
[21] Spotify. Blog de desarrolladores. https://labs.spotify.com/
[22] Spotify. Commoditizing Music Machine Learning: Services. https://labs.spotify.com/2016/08/07/commodity-music-ml-services/
[23] SoundCloud. https://soundcloud.com/
[24] SoundCloud. Blog de desarrolladores. Arquitectura. https://developers.soundcloud.com/blog/category/architecture

Glosario

API	Application Programming Interface
Endpoint	Punto de entrada a un servicio, proceso, o recurso de una arquitectura orientada a servicios.
DASH	Dynamic Adaptative Streaming over HTTP. También conocido como MPEG-DASH.

Anexos

A Manual de instalación

Para el cliente solo necesitamos Node.js, para el servidor Node.js, MongoDB, Redis, FFmpeg y GPAC (para MP4Box). Lo primero que haremos será instalar todo este software. Lo más recomendable es visitar las páginas oficiales de cada uno y mirar la documentación. Aun así, voy a mostrar cómo se instalan en Ubuntu 16.04. Todos pueden instalarse bajándose los archivos del servidor de la página con la documentación. También pueden ser instalados vía el gestor de paquetes apt.

Node.js

```
curl -sL https://deb.nodesource.com/setup_7.x | sudo -E bash -
sudo apt-get install -y nodejs
sudo apt-get install -y build-essential
```

MongoDB

```
echo "deb [ arch=amd64,arm64 ] http://repo.mongodb.org/apt/ubuntu xenial/mongodb-org/3.4
multiverse" | sudo tee /etc/apt/sources.list.d/mongodb-org-3.4.list

sudo apt-get update

sudo apt-get install -y mongodb-org

sudo service mongod start
```

Redis

```
sudo apt install redis-server
```

FFmpeg

```
sudo apt install ffmpeg
```

GPAC

```
sudo apt install gpac
```

Una vez que tenemos instalado todos los programas anteriores, podemos bajarnos todos el código del repositorio de git.

```
git clone https://adrian-bueno@bitbucket.org/adrian-bueno/tfg.git
```

Después entramos en el directorio /client e instalamos las dependencias de la aplicación.

```
npm install
```

Hacemos lo mismo para el directorio /server.

B Manual del programador

Ahora que ya tenemos instalado todo lo que necesitamos vamos a proceder a configurar las aplicaciones.

Será necesario tener activa la base de datos MongoDB y la base de datos Redis para ejecutar el servidor.

Para el servidor, en el fichero **src/global.ts** podemos cambiar el puerto, la IP, la dirección de la base de datos de MongoDB y los directorios en los que se guardan las canciones y las imágenes. La IP el puerto los utilizo también para crear una variable baseUrl que utilizó al devolver los JSON a los clientes. Se podría utilizar alguna dependencia más para obtener la IP pero he preferido ponerla a mano. Por defecto la IP tiene el valor 127.0.0.1, para acceder correctamente al servidor desde fuera del host en el que se encuentra, cambiar la IP a la IP pública.

Antes de iniciar el servidor debemos ejecutar el comando '**npm run build**' con el que se generará un directorio **dist** al que irán todos los archivos compilados y las carpetas public y views. Si se cambia solo un archivo TypeScript se puede ejecutar solo el compilador de TypeScript, '**npm run tsc**'. Una vez generado el directorio dist, utilizamos el comando '**npm start**' para iniciar el servidor.

Antes de ejecutar el cliente y con el servidor levantado, accedemos a la url: http://127.0.0.1:8080/oauth2/authorize. Donde nos aparece la pantalla de login y register que se muestra en la figura 3-2. Después de registrar un usuario nos aparecerá un error, que se debe a que nos faltaban argumentos en la URL, pero ahora eso no nos interesa. Haciendo esto se habrá creado un nuevo usuario en la base de datos, utilizando la consola de MongoDB o el endpoint de pruebas **/all/users** cogeremos el id de este usuario recién creado. Luego necesitaremos registrar un cliente, utilizaremos algún comando como **curl** para hacer una petición al servidor enviando un JSON con el nombre del cliente y el id del usuario que la quiere crear.

```
curl -H "Content-Type: application/json" -X POST -d '{"name": "MyClientName", "userId": "592c47e89cb65c1cf265f5e7"}' http://127.0.0.1:8080/clients
```

Nos responderá con un JSON que contiene el ID y la clave secreta del cliente:

```
{"message":"Client saved correctly.",
"data":{"name":"MyClientName","secret":"gbcg8sit8ovl8gn0ol253q3f5olvtbpex2gmooa6am0h4t835aupmpfvibaag72uttiklxuouy7proys21v6ffvmfdu8a4rasfqom7bn9o5wv80royjk4pr3q5fwcir5","userId":"592c47e89cb65c1cf265f5e7","id":"592c48e3284ad31cda38706c"}}
```

Ahora tendremos que ir al fichero **client/src/app/services/bluzu.service.ts** y cambiar el valor de las variables **clientId** y **clientSecret**. Después de realizar todo esto, podemos ejecutar el comando '**npm start**' para iniciar la aplicación con el servidor de desarrollo o hacer la compilación AOT y dejar a Nginx que se encargue de servir los archivos de la aplicación.

C Producción

Para el servidor no necesitamos realizar nada, como esta, está listo para producción.

Para el cliente haciendo '**npm start**' estaremos ejecutando un servidor pensado para el desarrollo. Lo que se recomienda en la documentación de Angular es realizar una compilación AOT (Ahead-of-Time) de la aplicación. Al hacer esta compilación tendremos en un solo fichero JavaScript, todos los archivos del directorio **/app**. Este fichero está lo mas comprimido posible y la diferencia de carga entre un solo archivo más grande y muchos pequeños es increíblemente grande. La aplicación siendo servida y pedida en local, sin compilar tarda al menos 1 segundo en cargar, compilado carga casi de forma instantánea. El JavaScript generado de nuestra aplicación ocupa unos 633 KB.

He creado un script **build-prod-browser.sh** para hacer el proceso de compilación automático. También se han definido los scripts **aot** y **rollup** en **package.json**. Para realizar la compilación AOT también tenemos que hacer una copia de varios archivos y hacer unas modificaciones, estos archivos son tsconfig-aot.json, src/index-aot.html, src/main-aot.ts y otro fichero de configuración rollup.config.js. En el index-aot.html también cogemos los scripts desde varios CDN en vez de desde nuestro servidor.

El código generado por el script **build-prod-browser.sh** estará en una carpeta **/dist**.He utilizado Nginx como servidor estático para servir el contenido de esta carpeta de forma rápida.

Nota: En el código veréis que se soporta Electron, para poder ejecutar la aplicación como una aplicación de escritorio. He tenido que comentar unas partes del código en las clases de Angular porque la compilación AOT no funciona si no, haciendo que no funcione bien la aplicación desde Electron hasta que se descomente el código comentado.

D Código

El código se encuentra en un repositorio público de git. El código está licenciado bajo la licencia MIT, por lo que puede ser usado para lo que se quiera, lo único que hay que cumplir en este tipo de licencia es indicar quien o quienes son los autores originales.

Repositorio: https://bitbucket.org/adrian-bueno/tfg

Servidor

package.json

```
{
 "name": "server",
 "version": "1.0.0",
 "description": "Servidor TFG",
 "main": "dist/server.js",
 "scripts": {
   "start": "node dist/server.js",
   "https": "node dist/server.js -https",
   "dev": "tsc && concurrently \"grunt\" \"tsc -w\" \"nodemon
dist/server.js\"",
   "forever": "forever start dist/server.js",
   "pm2": "pm2 start dist/server.js",
   "build": "grunt && tsc",
   "tsc": "tsc",
   "tsc:w": "tsc -w",
   "grunt": "grunt"
 },
 "author": "Adrian Bueno Jimenez",
 "license": "MIT",
 "dependencies": {
   "bcryptjs": "^2.3.0",
   "body-parser": "^1.15.2",
   "color-thief-jimp": "^2.0.2",
   "connect-ensure-login": "^0.1.1",
   "connect-redis": "^3.2.0",
   "ejs": "^2.5.2",
   "express": "^4.14.0",
   "express-session": "^1.14.2",
   "fs-extra": "^2.0.0",
   "jimp": "^0.2.27",
   "mediaserver": "^0.1.0",
   "mongoose": "^4.6.8",
   "morgan": "^1.7.0",
   "mp3-duration": "^1.0.10",
   "multer": "^1.2.0",
   "oauth2orize": "^1.5.1",
   "passport": "^0.3.2",
   "passport-http": "^0.3.0",
   "passport-http-bearer": "^1.0.1",
   "passport-local": "^1.0.0",
   "sharp": "^0.17.2",
   "uid": "0.0.2"
 },
```

```
 "devDependencies": {
   "@types/body-parser": "^1.16.3",
   "@types/debug": "^0.0.29",
   "@types/ejs": "^2.3.33",
   "@types/express": "^4.0.34",
   "@types/express-session": "0.0.32",
   "@types/mongoose": "^4.5.42",
   "@types/morgan": "^1.7.32",
   "@types/multer": "^0.0.33",
   "@types/node": "^7.0.0",
   "@types/oauth2orize": "^1.5.0",
   "@types/passport": "^0.3.3",
   "@types/passport-http": "^0.3.2",
   "@types/passport-http-bearer": "^1.0.30",
   "@types/passport-local": "^1.0.29",
   "concurrently": "^3.1.0",
   "forever": "^0.15.3",
   "grunt": "^1.0.1",
   "grunt-contrib-copy": "^1.0.0",
   "grunt-contrib-watch": "^1.0.0",
   "grunt-ts": "^6.0.0-beta.3",
   "nodemon": "^1.11.0",
   "pm2": "^2.2.2",
   "ts-node": "^3.0.2",
   "typescript": "^2.0.10"
 }
}
```

tsconfig.json

```
{
   "compilerOptions": {
       "target": "es2015",
       "module": "commonjs",
       "moduleResolution": "node",
       "sourceMap": true,
       "emitDecoratorMetadata": true,
       "experimentalDecorators": true,
       "removeComments": true,
       "noImplicitAny": false,
       "rootDir": "src",
       "outDir": "dist",
       "lib": [ "es2015", "dom" ]
   }
}
```

server.ts

```
import * as http    from "http";
import * as https   from "https";
import * as fs      from "fs";
import * as cluster from "cluster";
import * as os      from "os";

import * as global from "./global";
import app         from "./app";

const numCPUs = os.cpus().length;

// Check parameters
// Only one optional parameter is accepted: -https
if (process.argv.length > 3 ||
   (process.argv.length == 3 && process.argv[2] != "-https")) {
   console.log("\x1b[1;31m" + "\nThere is only one optional parameter: -https"
+ "\x1b[0m");
   process.exit(1);
}

// For HTTPS (argv[2] == "-https")
let options;
if (process.argv.length == 3) {
   options = {
       key: fs.readFileSync("dist/openssl/key.pem"),
       cert: fs.readFileSync("dist/openssl/server.crt")
   }
}

let server;
const port = normalizePort(process.env.PORT || global.port);

if (cluster.isMaster) {
   console.log(`Master ${process.pid} is running`);

   // Fork workers
   for (let i = 0; i < numCPUs; i++) {
       cluster.fork();
   }

   cluster.on("exit", (worker, code, signal) => {
       console.log(`Worker ${worker.process.pid} died`);
   });
} else {
   // Workers can share any TCP connection
   // In this case it is an HTTP server
```

```
    if (options)
        server = https.createServer(options, app);
    else
        server = http.createServer(app);

    server.listen(port);

    // Event functions
    server.on('error', onError);
    server.on('listening', onListening);

    // http.createServer(app).listen(port);
    console.log(`Worker ${process.pid} started`);
}

function normalizePort(val: number | string): number | string | boolean {
    let port: number = (typeof val === 'string') ? parseInt(val, 10) : val;
    if (isNaN(port)) return val;
    else if (port >= 0) return port;
    else return false;
}

function onError(error: NodeJS.ErrnoException): void {

    if (error.syscall !== 'listen') throw error;
    let bind = (typeof port === 'string') ? 'Pipe ' + port : 'Port ' + port;

    switch (error.code) {
        case 'EACCES':
            console.error(`${bind} requires elevated privileges`);
            process.exit(1);
            break;
        case 'EADDRINUSE':
            console.error(`${bind} is already in use`);
            process.exit(1);
            break;
        default:
            throw error;
    }
}

function onListening(): void {
    let addr = server.address();
    let bind = (typeof addr === 'string')
        ? `pipe ${addr}`
        : `port ${addr.port}`;

    console.log(`Listening on ${bind}`);
```

```
}
```

app.ts

```
import * as path       from 'path';
import * as express    from 'express';
import * as session    from 'express-session';
import * as logger     from 'morgan';
import * as bodyParser from 'body-parser';
import * as ejs        from 'ejs';
import * as ConnectRedis from 'connect-redis';
let RedisStore = ConnectRedis(session);

import {
   AllRouter,
   ClientsRouter,
   MeRouter,
   OAuth2Router,
   PlaylistRouter,
   SearchRouter,
   StreamRouter,
   TracksRouter,
   UsersRouter
} from "./routes";

/**
* Creates and configures an ExpressJS web server.
*/
class App {

   // Ref. to Express instance
   express: express.Application;

   /**
    * Run configuration methods on the Express instance.
    */
   constructor() {
       this.express = express();
       this.middleware();
       this.routes();
   }

   /**
    * Configure Express middleware.
    */
   private middleware(): void {
       this.express.use(logger('dev'));
```

```
        this.express.use(bodyParser.json());
        this.express.use(bodyParser.urlencoded({ extended: true }));

        // Static files
        this.express.use(express.static(__dirname + '/public'));
        this.express.use("/images", express.static(path.join(__dirname,
"../storage/images")));
        this.express.use("/audio", express.static(path.join(__dirname,
"../storage/audio")));

        // View Engine
        this.express.set("views", path.join(__dirname, "views"));
        this.express.set("view engine", "ejs");
        this.express.engine("ejs", ejs.renderFile);

        // Use express session support since OAuth2orize requires it
        // Use Redis to store sessions (necessary if using processes cluster)
        this.express.use(session({
            store: new RedisStore({ttl: 300}), // max session live = 5 min
            secret: "Super Secret Session Key"
        }));

        // Headers, allow CORS
        this.express.use((req, res, next) => {
            res.header("Access-Control-Allow-Origin", "*");
            res.header("Access-Control-Allow-Methods", "GET, POST, PUT,
DELETE");
            res.header("Access-Control-Allow-Headers", "Origin, X-Requested-
With, Content-Type, Accept, Authorization");
            next();
        });
    }

    /**
     * Configure API endpoints.
     */
    private routes(): void {

        let RootRouter = express.Router();
        RootRouter.get('/', (req, res, next) => {
            res.json({
                message: 'Welcome to our API!'
            });
        });

        this.express.use('/'         , RootRouter);
        this.express.use('/all'      , AllRouter);
        this.express.use('/clients'  , ClientsRouter);
```

```
        this.express.use('/me'       , MeRouter);
        this.express.use('/oauth2'   , OAuth2Router);
        this.express.use('/playlists', PlaylistRouter);
        this.express.use('/search'   , SearchRouter);
        this.express.use('/stream'   , StreamRouter);
        this.express.use('/tracks'   , TracksRouter);
        this.express.use('/users'    , UsersRouter);
    }
}

export default new App().express;
```

database.ts

```
import * as mongoose from 'mongoose';

/**
* Initizalizes a connection with MongoDB.
* Returns db connection.
*/
export function databaseConnect(dbUrl: string): any {
    mongoose.connect(dbUrl);

    mongoose.connection.on("error", () => {
        console.log("\x1b[1;31m" + "Connection to database : ERROR" +
"\x1b[0m");
    });

    mongoose.connection.once("open", () => {
        console.log("Connection to database : OK");
    });

    return mongoose.connection;
}
```

global.ts

```
import { databaseConnect } from "./database";
import * as path           from "path";

// Server
export const ip      : string = "127.0.0.1";
export const port    : number = 8080;
export const baseUrl : string = `http://${ip}:${port}/`;

// Database
```

```
export const dbUrl : string = "mongodb://127.0.0.1/bluzu";
export const db    : any    = databaseConnect(dbUrl);

// Storage location
export const audioStoragePath : string = path.join(__dirname,
"../storage/audio/");
export const imageStoragePath : string = path.join(__dirname,
"../storage/images/");

// Dash script location
export const dashScript : string = path.join(__dirname, "generate-dash.sh");
```

generate-dash.sh

```
#!/bin/bash

bitrates=()
outputFiles=()
dir=$1
inidir=$PWD

cd $dir
mkdir "dash"

# Get audio bit rate
bit_rate=$(ffprobe -v quiet -print_format json -show_format original.mp3 | grep
bit_rate | grep -o '[0-9]\+')

if (( $bit_rate >= 128000 ))
then
   bitrates+=("128k")
   outputFiles+=("128.m4a")
fi

if (( $bit_rate >= 192000 ))
then
   bitrates+=("192k")
   outputFiles+=("192.m4a")
fi

if (( $bit_rate >= 256000 ))
then
   bitrates+=("256k")
   outputFiles+=("256.m4a")
fi

if (( $bit_rate >= 320000 ))
```

```
then
   bitrates+=("320k")
   outputFiles+=("320.m4a")
fi

max=${#bitrates[@]}
max=$((max - 1))

for i in `seq 0 ${max}`
do
   echo "[ffmpeg] Converting to" ${outputFiles[$i]}
   ffmpeg -y -loglevel panic -i original.mp3 -vn -c:a aac -strict -2 -b:a
${bitrates[$i]} ${outputFiles[$i]}
done

MP4Box -dash 4000 -frag 4000 -rap -segment-name segment_%s_ -out dash/index
${outputFiles[@]}

# Delete files generated by ffmpeg
for f in ${outputFiles[@]}
do
   rm -f $f
done

cd $inidir
```

models/user.ts

```
import { Schema, Types, model } from 'mongoose';
import { baseUrl }              from "../global";
let bcrypt = require("bcryptjs");

let UserSchema: any = new Schema({
   username        : { type: String, unique: true, required: true },
   email           : { type: String, required: true },
   profileImage    : { type: String, default: "images/default/no-profile.jpg"
},
   backgroundImage : { type: String, default: "images/default/no-
background.jpg" },
   name            : { type: String, required: true },
   password        : { type: String, required: true },
   public          : { type: Boolean, default: true },
   country         : String,
   description     : String,
});

UserSchema.set('toJSON', {
   transform: function (doc, ret, options) {
```

```
        ret.id = ret._id;
        ret.type = "user";
        if (ret.profileImage) ret.profileImage = baseUrl + ret.profileImage;
        if (ret.backgroundImage) ret.backgroundImage = baseUrl +
ret.backgroundImage;

        delete ret._id;
        delete ret.__v;
        delete ret.password;
    }
});

/**
* Execute before each User.save() call
*/
UserSchema.pre("save", function(callback) {

    let user = this;

    // Break out if the password hasn't changed
    if (!user.isModified("password")) {
        return callback();
    }

    // Password changed so we need to hash it
    bcrypt.hash(user.password, 5, (err, hash) => {
        if (err) {
            return callback(err);
        }

        user.password = hash;
        callback();
    });
});

export let User = model("User", UserSchema);
```

routes/stream.router.ts

```
import { Router, Request, Response, NextFunction } from "express";
import * as path   from "path";
import * as fs     from "fs";
let mediaserver = require("mediaserver");

import { audioStoragePath } from "../global";
import { Playlist, Track, User } from "../models";

class StreamRouterConfig {
```

```
    router: Router;

    constructor() {
        this.router = Router();
        this.routes();
    }

    routes() {
        this.router.get('/:id/dash/:file', this.streamDash);
        this.router.get('/:id/html5', this.streamHtml5);
    }

    streamHtml5(req: Request, res: Response, next: NextFunction) {
        let filePath: string =
${audioStoragePath}/${req.params.id}/original.mp3`;
        fs.stat(filePath, (err, stat) => {
            if (err) {
                return res.status(404).json({ err: "File does not exist." });
            }
            mediaserver.pipe(req, res, filePath);
        });
    }

    streamDash(req: Request, res: Response, next: NextFunction) {
        let filePath: string = path.join(audioStoragePath, req.params.id,
"dash",              req.params.file);

        fs.stat(filePath, (err, stat) => {
            if (err) {
                return res.status(404).json({ err: "File does not exist." });
            }
            let readStream = fs.createReadStream(filePath);
            readStream.pipe(res);
        });
    }
}

export let StreamRouter = new StreamRouterConfig().router;
```

routes/all.router.ts

```
import { Router, Request, Response, NextFunction } from "express";
import { Types } from "mongoose";

import { Playlist, Track, User } from "../models";
```

```
class AllRouterConfig {

   router: Router;

   constructor() {
       this.router = Router();
       this.routes();
   }

   routes() {
       this.router.get('/playlists', this.getAllPlaylists);
       this.router.get('/users', this.getAllUsers);
       this.router.get('/tracks', this.getAllTracks);
   }


    getAllPlaylists(req: Request, res: Response, next: NextFunction) {
        Playlist.find((err, playlists) => {
            if (err)
                res.status(500).json({ message: "Error getting playlists." });
            else
                res.status(200).json(playlists);
        }).populate("owner", "username name");
    }

   getAllUsers(req: Request, res: Response, next: NextFunction) {
       User.find((err, users) => {
           if (err)
               res.status(500).json({ message: "Error getting users." });
           else
               res.status(200).json(users);
       });
   }

   getAllTracks(req: Request, res: Response, next: NextFunction) {
       Track.find((err, tracks) => {
           if (err)
               res.status(500).json({ message: "Error getting tracks." });
           else
               res.status(200).json(tracks);
       }).populate("owner", "username name");;
   }

}

export let AllRouter = new AllRouterConfig().router;
```

Cliente

package.json

```
{
 "name": "client",
 "productName": "ClienteTFG",
 "version": "1.0.0",
 "description": "Cliente TFG hecho con Angular. Ejecutable a traves de Electron",
 "main": "src/electron.js",
 "scripts": {
   "start": "tsc && concurrently \"tsc -w\" \"lite-server -c=lite-server.config.json\" ",
   "lite": "lite-server -c=lite-server.config.json",
   "tsc": "tsc",
   "tsc:w": "tsc -w",
   "aot": "ngc -p tsconfig-aot.json",
   "rollup": "rollup -c rollup.config.js",
   "build-prod": "ngc -p tsconfig-aot.json && rollup -c rollup.config.js",
   "build-win64": "build --win --x64"
 },
 "author": "Adrian Bueno Jimenez",
 "license": "MIT",
 "dependencies": {
   "@angular/animations": "^4.0.1",
   "@angular/common": "^4.0.1",
   "@angular/compiler": "^4.0.1",
   "@angular/core": "^4.0.1",
   "@angular/forms": "^4.0.1",
   "@angular/http": "^4.0.1",
   "@angular/platform-browser": "^4.0.1",
   "@angular/platform-browser-dynamic": "^4.0.1",
   "@angular/router": "^4.0.1",
   "angular-in-memory-web-api": "^0.3.0",
   "async": "^2.1.5",
   "core-js": "^2.4.1",
   "dashjs": "^2.4.0",
   "hammer-time": "^2.0.0",
   "hammerjs": "^2.0.8",
   "jquery": "^3.2.0",
   "material-design-icons": "^3.0.1",
   "materialize-css": "^0.98.0",
   "rxjs": "^5.0.3",
   "systemjs": "^0.20.0",
   "zone.js": "^0.8.5"
 },
 "devDependencies": {
   "@angular/compiler-cli": "^4.0.1",
   "@angular/platform-server": "^4.0.1",
   "@types/core-js": "^0.9.35",
   "@types/electron": "^1.4.31",
   "@types/hammerjs": "^2.0.34",
```

```
    "@types/jquery": "^2.0.41",
    "@types/materialize-css": "^0.98.0",
    "@types/node": "^7.0.0",
    "concurrently": "^3.1.0",
    "electron": "^1.4.14",
    "electron-builder": "^11.2.4",
    "lite-server": "^2.2.2",
    "rollup": "^0.41.6",
    "rollup-plugin-commonjs": "^8.0.2",
    "rollup-plugin-node-resolve": "^3.0.0",
    "rollup-plugin-uglify": "^1.0.1",
    "source-map-explorer": "^1.3.3",
    "typescript": "^2.2.2"
  },
  "build": {
    "appId": "com.adrian-bueno.clienttfg",
    "copyright": "Copyright (c) 2016-2017 Adrian Bueno",
    "mac": {},
    "win": {},
    "linux": {}
  }
}
```

tsconfig.json

```
{
    "compilerOptions": {
        "target": "es2015",
        "module": "commonjs",
        "moduleResolution": "node",
        "sourceMap": true,
        "emitDecoratorMetadata": true,
        "experimentalDecorators": true,
        "removeComments": false,
        "noImplicitAny": false,
        "lib": ["es2015", "dom"]
    },
    "exclude": [
        "node_modules",
        "aot",
        "src/main-aot.ts"
    ]
}
```

src/systemjs.config.js

```
(function (global) {
  System.config({
    paths: {
      // paths serve as alias
      'npm:': 'node_modules/'
    },
    // map tells the System loader where to look for things
```

```
   map: {
     // our app is within the app folder
     'app': 'app',

     // angular bundles
     '@angular/core': 'npm:@angular/core/bundles/core.umd.js',
     '@angular/common': 'npm:@angular/common/bundles/common.umd.js',
     '@angular/compiler': 'npm:@angular/compiler/bundles/compiler.umd.js',
     '@angular/platform-browser': 'npm:@angular/platform-browser/bundles/platform-
browser.umd.js',
     '@angular/platform-browser-dynamic': 'npm:@angular/platform-browser-
dynamic/bundles/platform-browser-dynamic.umd.js',
     '@angular/http': 'npm:@angular/http/bundles/http.umd.js',
     '@angular/router': 'npm:@angular/router/bundles/router.umd.js',
     '@angular/forms': 'npm:@angular/forms/bundles/forms.umd.js',

     // other libraries
     'rxjs':                      'npm:rxjs',
     'angular-in-memory-web-api': 'npm:angular-in-memory-web-api/bundles/in-memory-web-
api.umd.js'
   },
   // packages tells the System loader how to load when no filename and/or no extension
   packages: {
     app: {
       defaultExtension: 'js',
       meta: {
         './*.js': {
           loader: 'systemjs-angular-loader.js'
         }
       }
     },
     rxjs: {
       defaultExtension: 'js'
     }
   }
 });
})(this);
```

src/main.ts

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { enableProdMode } from '@angular/core';

import { AppModule } from './app/app.module';

// Enable production mode unless running locally
if (!/localhost/.test(document.location.host))
   enableProdMode();

platformBrowserDynamic().bootstrapModule(AppModule);
```

src/index.html

```
<!DOCTYPE html>
<html>
<head>
   <base id="base" href="/"> <!-- Use "/" for browser and "./" for electron -->
   <title>Bluzu</title>
   <meta charset="UTF-8">
   <meta name="viewport" content="width=device-width, initial-scale=1">
   <meta name="theme-color" content="#212121"> <!-- Chrome, Firefox OS and Opera -->
   <meta name="msapplication-navbutton-color" content="#212121"> <!-- Windows Phone -->
   <meta name="apple-mobile-web-app-status-bar-style" content="#212121"> <!-- iOS Safari
-->
   <link rel="manifest" href="manifest.json">

   <!-- Stylesheets -->
   <link rel="stylesheet" href="node_modules/materialize-
css/dist/css/materialize.min.css">
   <link rel="stylesheet" href="node_modules/material-design-icons/iconfont/material-
icons.css">
   <link rel="stylesheet" href="css/loading.css">
   <link rel="stylesheet" href="css/custom.css">
</head>

<body>
   <div class="app-container">
       <app>
           <div class="center">
               <div class="loader"></div>
           </div>
       </app>
   </div>

   <!-- Polyfills for older browsers -->
   <script src="node_modules/core-js/client/shim.min.js"></script>

   <script src="node_modules/zone.js/dist/zone.js"></script>
   <script src="node_modules/reflect-metadata/Reflect.js"></script>
   <script src="node_modules/systemjs/dist/system.src.js"></script>

   <!-- Configure SystemJS -->
   <script src="systemjs.config.js"></script>
   <script>
       System.import('main.js').catch(function(err) {
           console.error(err);
       });
   </script>

   <script src="js/index.js"></script>
   <script src="node_modules/dashjs/dist/dash.all.min.js"></script>
   <script src="node_modules/jquery/dist/jquery.min.js"></script>
   <script src="node_modules/hammerjs/hammer.min.js"></script>
   <script src="node_modules/materialize-css/dist/js/materialize.min.js"></script>
</body>
</html>
```

src/app/app.module.ts

```
import { NgModule }      from '@angular/core';
import { BrowserModule, HammerGestureConfig, HAMMER_GESTURE_CONFIG } from
'@angular/platform-browser';
import { FormsModule }   from '@angular/forms';
import { HttpModule }    from '@angular/http';

import { InViewportModule } from './modules/in-viewport/index';

import { AppComponent }  from './app.component';
import { AppRoutingModule } from './app.routing';
import {
   SearchComponent,
   SidenavComponent,
   SidenavNoLoggedComponent,
   SidenavOptionComponent,
   PlayQueueComponent,
   PlayerComponent,
   PlayerBigComponent,
   PlaylistComponent,
   HomeComponent,
   UploadComponent,
   UploadTrackComponent,
   UserComponent,
   UserHomeComponent,
   UserMusicComponent,
   UserPlaylistsComponent,
   UserSimilarArtistsComponent,
   UserAboutComponent,
   BoxComponent,
   ContextMenuComponent,
   EditPlaylistFormComponent,
} from './components';

import { BluzuService, GlobalService, PlayerService } from './services';

/**
* Hammer touch events custom configuration
*/
export class HammerConfig extends HammerGestureConfig {
   overrides = <any> {
       'swipe': { direction: Hammer.DIRECTION_ALL, velocity: 0.3, threshold: 10},
       'pan': { direction: Hammer.DIRECTION_ALL, velocity: 0.3, threshold: 10 }
   }
}

@NgModule({
   imports: [
       BrowserModule,
       FormsModule,
       HttpModule,
       AppRoutingModule,
       InViewportModule.forRoot()
```

```
    ],
    declarations: [
        AppComponent,
        SearchComponent,
        SidenavComponent,
        SidenavNoLoggedComponent,
        SidenavOptionComponent,
        PlayQueueComponent,
        PlayerComponent,
        PlayerBigComponent,
        PlaylistComponent,
        HomeComponent,
        UploadComponent,
        UploadTrackComponent,
        UserComponent,
        UserHomeComponent,
        UserMusicComponent,
        UserPlaylistsComponent,
        UserSimilarArtistsComponent,
        UserAboutComponent,
        BoxComponent,
        ContextMenuComponent,
        EditPlaylistFormComponent,
    ],
    providers: [
        BluzuService,
        GlobalService,
        PlayerService,
        {
            provide: HAMMER_GESTURE_CONFIG,
            useClass: HammerConfig
        }
    ],
    bootstrap: [AppComponent]
})

export class AppModule { }
```

src/app/app.routing.ts

```
import { NgModule }  from '@angular/core';
import { Routes, RouterModule } from '@angular/router';

import {
    HomeComponent,
    PlaylistComponent,
    SearchComponent,
    UploadComponent,
    UserComponent,
    UserHomeComponent,
    UserMusicComponent,
    UserPlaylistsComponent,
    UserSimilarArtistsComponent,
```

```
   UserAboutComponent
} from './components';

const appRoutes: Routes = [
   {
       path: 'search',
       component: SearchComponent
   },
   {
       path: 'playlist/:id',
       component: PlaylistComponent
   },
   {
       path: 'upload',
       component: UploadComponent
   },
   {
       path: 'user/:id',
       component: UserComponent,
       children: [
           {
               path: '',
               redirectTo: 'home',
               pathMatch: 'full'
           },
           {
               path: 'home',
               component: UserHomeComponent
           },
           {
               path: 'music',
               component: UserMusicComponent
           },
           {
               path: 'playlists',
               component: UserPlaylistsComponent
           },
           {
               path: 'similar-artists',
               component: UserSimilarArtistsComponent
           },
           {
               path: 'about',
               component: UserAboutComponent
           },
       ]
   },
   {
       path: '',
       component: HomeComponent
   },
   {
       path: '**',
       redirectTo: '/',
```

```
    },
];

@NgModule({
    imports: [
        RouterModule.forRoot(appRoutes)
    ],
    exports: [
        RouterModule
    ]
})

export class AppRoutingModule {}
```

src/app/components/search/search.component.ts

```
import { Component } from '@angular/core';
import { Playlist, User, Track } from '../../models';
import { BluzuService } from '../../services';

@Component({
    moduleId: module.id,
    selector: 'search',
    templateUrl: 'search.component.html',
    styleUrls: ['search.component.css']
})

export class SearchComponent {

    playlists: Playlist[] = [];
    users: User[] = [];
    tracks: Track[] = [];
    searchInput: string;

    constructor(private bluzuService: BluzuService) { }

    search(searchString: string) {
        if (searchString === "") {
            this.playlists = [];
            this.users = [];
            this.tracks = [];
            return;
        }

        this.bluzuService.searchPlaylists(searchString)
            .subscribe(playlists => this.playlists = playlists);

        this.bluzuService.searchUsers(searchString)
            .subscribe(users => this.users = users);

        this.bluzuService.searchTracks(searchString)
            .subscribe(tracks => this.tracks = tracks);
    }
```

```
}
```

src/app/components/search/search.component.html

```
<nav>
   <div class="nav-wrapper">
       <form>
           <div class="input-field">
               <input id="search" type="search" placeholder="Search" required
                   name="search" [(ngModel)]="searchInput"
                   (ngModelChange)="search(searchInput)">
               <label for="search"><i class="material-icons">search</i></label>
           </div>
       </form>
   </div>
</nav>

<div id="search-output">
   <div *ngIf="playlists.length !== 0" class="divider">
       <div class="title">PLAYLISTS</div>
   </div>
   <box *ngFor="let playlist of playlists" [object]="playlist"></box>

   <div *ngIf="users.length !== 0" class="divider">
       <div class="title">USERS</div>
   </div>
   <box *ngFor="let user of users" [object]="user"></box>

   <div *ngIf="tracks.length !== 0" class="divider">
       <div class="title">TRACKS</div>
   </div>
   <box *ngFor="let track of tracks" [object]="track"></box>
</div>
```

src/app/services/bluzu.service.ts

Los tres puntos suspensivos indican que hay más código, pero no se ha incluido.

```
import { Injectable, Output, EventEmitter  } from '@angular/core';
import { Http, Headers } from '@angular/http';
import { Observable } from 'rxjs';
import 'rxjs/add/operator/map';
import 'rxjs/add/operator/toPromise';
import { Playlist, User, Track } from '../models';

@Injectable()
export class BluzuService {

   loggedUser: User;
   accessToken: string;
   bluzuApiBaseUrl: string = "http://192.168.1.109:8080";
   baseUrl: string = "http://192.168.1.109";
```

```
    clientId: string = "585f3518792fc91038361a29";
    clientSecret: string = "this_is_my_secret";
    redirectUri: string = encodeURI(`${this.baseUrl}/`);
    authorizeUri: string =
`${this.bluzuApiBaseUrl}/oauth2/authorize?client_id=${this.client_Id}&response_type=code&
redirect_uri=${this.redirectUri}`;

    @Output() logEmitter$: EventEmitter<User> = new EventEmitter();
    @Output() newTrack$: EventEmitter<Track> = new EventEmitter();
    @Output() deletedTrack$: EventEmitter<string> = new EventEmitter();
    @Output() newPlaylist$: EventEmitter<Playlist> = new EventEmitter();
    @Output() deletedPlaylist$: EventEmitter<string> = new EventEmitter();

    constructor(private http: Http) { ... }

    ...

    addTrackToPlaylist(trackId: string, playlistId: string): Promise<Playlist> {
        let headers: Headers = new Headers();
        headers.append("content-type", "application/json");
        headers.append("authorization", `Bearer ${this.accessToken}`);

        let body = {
            trackId: trackId,
            playlistId: playlistId
        };

        return this.http.put(`${this.bluzuApiBaseUrl}/me/playlist/${playlistId}/add-
track`, body, { headers: headers })
            .map((res:any) => {
                Materialize.toast(res.json().message, 2500, "blue-bg rounded");
                return res.json().playlist;
            }).toPromise()
            .catch((err:any) => {
                Materialize.toast(err.json().message, 3000, "red-bg rounded");
                return {};
            });
    }
    ...

    searchTracks(searchString: string): Observable<Track[]> {
        return this.http.get(`${this.bluzuApiBaseUrl}/search/tracks/${searchString}`)
            .map(res => res.json());
    }

    searchPlaylists(searchString: string): Observable<Playlist[]> {
        return this.http.get(`${this.bluzuApiBaseUrl}/search/playlists/${searchString}`)
            .map(res => res.json());
    }

    searchUsers(searchString: string): Observable<User[]> {
        return this.http.get(`${this.bluzuApiBaseUrl}/search/users/${searchString}`)
            .map(res => res.json());
    }
```

```
   ...
}
```

E Ejemplo index.mpd generado por MP4Box

```
<?xml version="1.0"?>
<!-- MPD file Generated with GPAC version 0.5.2-DEV-revVersion: 0.5.2-426-gc5ad4e4+dfsg5-
1build1  at 2017-03-26T16:41:14.670Z-->
<MPD xmlns="urn:mpeg:dash:schema:mpd:2011" minBufferTime="PT1.500S" type="static"
mediaPresentationDuration="PT0H3M17.774S" maxSegmentDuration="PT0H0M3.993S"
profiles="urn:mpeg:dash:profile:full:2011">
<ProgramInformation moreInformationURL="http://gpac.sourceforge.net">
 <Title>dash/index.mpd generated by GPAC</Title>
</ProgramInformation>

<Period duration="PT0H3M17.774S">
 <AdaptationSet segmentAlignment="true" bitstreamSwitching="true" lang="und">
  <AudioChannelConfiguration
schemeIdUri="urn:mpeg:dash:23003:3:audio_channel_configuration:2011" value="2"/>
  <SegmentList>
   <Initialization sourceURL="index_init.mp4"/>
  </SegmentList>
  <Representation id="1" mimeType="audio/mp4" codecs="mp4a.40.2"
audioSamplingRate="44100" startWithSAP="1" bandwidth="130507">
   <SegmentList timescale="44100" duration="176091">
    <SegmentURL media="segment_128_1.m4s"/>
    <SegmentURL media="segment_128_2.m4s"/>
    <SegmentURL media="segment_128_3.m4s"/>
    ...
   <SegmentURL media="segment_128_50.m4s"/>
<SegmentURL media="segment_128_49.m4s"/>
    <SegmentURL media="segment_128_50.m4s"/>
   </SegmentList>
  </Representation>
  <Representation id="2" mimeType="audio/mp4" codecs="mp4a.40.2"
audioSamplingRate="44100" startWithSAP="1" bandwidth="192625">
   <SegmentList timescale="44100" duration="176091">
    <SegmentURL media="segment_192_1.m4s"/>
    <SegmentURL media="segment_192_2.m4s"/>
    <SegmentURL media="segment_192_3.m4s"/>
    ...
    <SegmentURL media="segment_192_50.m4s"/>
   </SegmentList>
  </Representation>
 </AdaptationSet>
</Period>
</MPD>
```